探索和品味Android大师们的内核设计艺术



张元亮◎编著

深入理解Android系统

- 全面剖析进程/线程、内存管理、Binder机制、显示系统、多媒体管理、输入系统等核心知识在Android中的实现原理。
- 源码分析+全真示例+图片解析=更易于理解的思维路径。
- 由浅入深，由总体框架到细节实现，快速获取对Android系统的二次开发能力。
- 教授精髓，精讲精炼。赠送源码，拿来就用。

清华大学出版社

深入理解 Android 系统

张元亮　编著

清华大学出版社
北　京

内 容 简 介

本书内容共 18 章，循序渐进地分析了整个 Android 系统的基本架构知识，从获取源码开始讲起，依次讲解了 Android 系统介绍，包括获取并编译 Android 源码，分析 JNI，内存系统架构详解，硬件抽象层架构详解，Binder 通信机制详解，init 启动进程详解，Zygote 进程详解，System 进程详解，应用程序进程详解，ART 机制架构详解，Sensor 传感器系统架构详解，蓝牙系统架构详解，Android 多媒体框架架构详解，音频系统框架架构详解，视频系统架构详解，WebKit 系统架构详解，Android 5.0 中的 WebView，Wi-Fi 系统架构详解等内容。本书几乎涵盖了所有 Android 系统架构的主要核心内容，讲解方法通俗易懂并且详细，不但适合应用高手们学习，也特别便于初学者学习和理解。

本书适合 Android 源码分析人员、Android 系统架构师、Linux 开发人员、Android 物联网开发人员、Android 爱好者、Android 底层开发人员、Android 驱动开发人员、Android 应用开发人员、Android 传感器开发人员、Android 智能家居开发人员、Android 可穿戴设备开发人员学习，也可以作为相关培训学校和大专院校相关专业的教学用书。

图书在版编目（CIP）数据

深入理解 Android 系统/张元亮编著．—北京：清华大学出版社，2015
ISBN 978-7-302-40439-2

I．①深…　II．①张…　III．①移动终端-应用程序-程序设计　IV．①TN929.53

中国版本图书馆 CIP 数据核字（2015）第 122705 号

责任编辑： 朱英彪
封面设计： 刘　超
版式设计： 魏　远
责任校对： 王　云
责任印制：

出版发行： 清华大学出版社
网　　址： http://www.tup.com.cn，http://www.wqbook.com
地　　址： 北京清华大学学研大厦 A 座　　**邮　　编：** 100084
社 总 机： 010-62770175　　**邮　　购：** 010-62786544
投稿与读者服务： 010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销： 全国新华书店
开　　本： 203mm×260mm　**印　　张：** 44.5　**字　　数：** 1248 千字
版　　次： 2015 年 7 月第 1 版　**印　　次：** 2015 年 7 月第 1 次印刷
印　　数： 1～3000
定　　价： 88.00 元

产品编号：061537-01

前　　言

2007 年 11 月 5 日，谷歌公司宣布基于 Linux 平台的开源手机操作系统 Android 诞生，该平台号称是首个为移动终端打造的真正开放和完整的移动软件平台。本书将和广大读者一起深入理解 Android 系统的架构知识，共同领略这款神奇系统的奥妙之处。

市场占有率高居第一

截至 2014 年 9 月，Android 在智能手机市场上的占有率从 2013 年第一季度的 68.8%上升到 85%。而 iOS 则从 2013 年的 19.4%下降到 15.5%，WP 系统从原来的 2.7%小幅上升到 3.6%。从数据上看，Android 平台占据了市场的主导地位。

由数据可知，iOS 有所下降，WP 市场小幅增长，但 Android 市场的占有率增加幅度较大。就目前来看，智能手机的市场已经饱和，大多数用户都在各个平台间转换。而就在这样一个市场上，Android 还增长了 10%左右的占有率确实不易。

为开发人员提供了发展的平台

（1）保证开发人员可以迅速向 Android 应用开发转型

Android 应用程序是使用 Java 语言开发的，只要具备 Java 开发基础，就能很快入门并掌握。作为单独的 Android 应用开发，对 Java 编程门槛的要求并不高，即使没有编程经验，也可以在突击学习 Java 之后学习 Android。另外，Android 完全支持 2D、3D 和数据库，并且和浏览器实现了集成。所以通过 Android 平台，程序员可以迅速、高效地开发出绚丽多彩的应用，例如常见的工具、管理软件、互联网应用和游戏等。

（2）定期举办奖金丰厚的 Android 开发大赛

为了吸引更多的用户使用 Android 开发，已经成功举办了奖金为数千万美元的开发者竞赛。鼓励开发人员创建出创意十足且实用的软件。对于开发人员来说，这种大赛不但能提升自己的开发水平，并且高额的奖金也是学员们学习的动力。

（3）开发人员可以利用自己的作品赚钱

为了能让 Android 平台引起更多的关注，谷歌提供了一个专门下载 Android 应用的门店 Android Market，地址是 https://play.google.com/store。在这个门店里面允许开发人员发布应用程序，也允许 Android 用户下载自己喜欢的程序。作为开发者，需要申请开发者账号，申请后才能将自己的程序上传到 Android Market，并且可以对自己的软件进行定价。只要所开发的软件程序足够吸引人，就可以获得很可观的金钱回报。这样实现了程序员学习和赚钱的两不误，所以吸引了更多开发人员加入到 Android 开发大军中来。

本书的内容

本书内容共 18 章，循序渐进地分析了整个 Android 系统的基本架构知识。本书从获取源码开始讲起，依次讲解了 Android 系统介绍，包括获取并编译 Android 源码，分析 JNI，内存系统架构详解，硬件抽象层架构详解，Binder 通信机制详解，init 启动进程详解，Zygote 进程详解，System 进程详解，应用程序进程详解，Sensor 传感器系统架构详解，蓝牙系统架构详解，Android 多媒体框架架构详解，音频系统框架架构详解，视频系统架构详解，WebKit 系统架构详解，Android 5.0 中的 WebView，Wi-Fi 系统架构详解，ART 机制架构详解等内容。本书几乎涵盖了所有 Android 系统架构的主要核心内容，讲解方法通俗易懂并且详细，不但适合应用高手们学习，也特别便于初学者学习和理解。

本书的版本

Android 系统自 2008 年 9 月发布第一个版本 1.1 以来，截至 2014 年 10 月发布最新版本 5.0，一共存在十多个版本。由此可见，Android 系统升级频率较快，一年之中最少有两个新版本诞生。如果过于追求新版本，会造成力不从心的结果。所以在此建议广大读者不必追求最新的版本，只需关注最流行的版本即可。据官方统计，截至 2014 年 11 月 10 日，占据前 3 位的版本分别是 Android 4.4、Android 4.3 和 Android 5.0，其实这 3 个版本的区别并不是很大，只是在某些领域的细节上进行了更新。为了在市场普及率和新版本之间做好兼顾，本书将以最新的 Android 5.0 作为讲解主线，并且结合了 Android 4.4 的架构知识。

本书特色

本书内容十分丰富，讲解细致。我们的目标是通过一本图书，提供多本图书的价值，读者可以根据自己的需要有选择地阅读。在内容的编写上，本书具有以下特色。

（1）内容全面，讲解细致

本书几乎涵盖了 Android 系统架构所需要的所有主要知识点，详细讲解了每一个 Android 系统的具体实现过程。每一个知识点都力求用详实易懂的语言展现在读者面前。

（2）遵循合理的主线进行讲解

为了使广大读者彻底弄清楚 Android 系统架构的各个知识点，在讲解每一个知识点时，从 Linux 内核开始讲起，依次剖析了底层架构、API 硬件抽象层和顶层应用的具体知识。遵循了从底层到顶层的顺序，实现了 Android 系统架构大揭秘的目标。

（3）章节独立，自由阅读

本书中的每一章内容都可以独自成书，读者既可以按照本书编排的章节顺序进行学习，也可以根据自己的需求对某一章节进行针对性的学习。并且和传统古板的计算机书籍相比，阅读本书会带来很大的快乐。

（4）版本新颖，代表性强

本书以最新的 Android 5.0 作为讲解主线，结合 Android 4.4 的架构知识进行讲解，这样可以涵盖大多数读者群体，代表性更强。

本书的读者对象

- ☑ Android 源码分析人员。
- ☑ Android 系统架构师。
- ☑ Linux 开发人员。
- ☑ Android 物联网开发人员。
- ☑ Android 爱好者。
- ☑ Android 底层开发人员。
- ☑ Android 驱动开发人员。
- ☑ Android 应用开发人员。
- ☑ Android 传感器开发人员。
- ☑ Android 智能家居开发人员。
- ☑ Android 可穿戴设备开发人员。
- ☑ 相关培训学校的学生。
- ☑ 相关大专院校的学生。

售后服务

参与本书编写的人员还有周秀、付松柏、邓才兵、钟世礼、谭贞军、张加春、王教明、万春潮、郭慧玲、侯恩静、程娟、王文忠、陈强、何子夜、李天祥、周锐、朱桂英、张元亮、张韶青、秦丹枫。本书在编写过程中，得到了清华大学出版社工作人员的大力支持，正是各位编辑的求实、耐心和效率，才使得本书在这么短的时间内出版。另外也十分感谢我的家人在我写作时给予的巨大支持。

由于编者水平有限，如有纰漏和不尽如人意之处，诚请读者提出意见或建议，以便修订并使之更臻完善。另外我们提供了售后支持网站（http://www.chubanbook.com/）和 QQ 群（192153124），读者朋友如有疑问可以在此提出，一定会得到满意的答复。

编　者

目　录



第 1 章　获取并编译 Android 源码

在分析 Android 源码之前，需要先获取 Android 系统的源码，并在自己的机器上进行编译。本章将详细讲解获取并编译 Android 5.0 源码的基本知识。另外，因为 Android 系统源码的文件数量巨大，目录结构层次复杂，所以将在本章对 Android 5.0 源码的目录结构进行整体分析，并详细介绍从 SDK 中生成 SDK 的方法。

1.1　获取 Android 源码

要想研究 Android 系统的源码，需要先获取其源码。目前，市面中的主流操作系统是 Windows、Linux 和 Mac OS。因为 Mac OS 属于类 Linux 系统，所以本书将讲解在 Windows 系统和 Linux 系统中获取 Android 源码的知识。

1.1.1　在 Linux 系统获取 Android 源码

在 Linux 系统中，通常使用 Ubuntu 来下载和编译 Android 源码。由于 Android 的源码内容很多，Google 采用了 git 的版本控制工具，并对不同的模块设置不同的 git 服务器，可以用 repo 自动化脚本来下载 Android 源码，下面逐步介绍如何获取 Android 源码的过程。

（1）下载 repo

在用户目录下，创建 bin 文件夹，用于存放 repo，并把该路径设置到环境变量中去，命令如下：

```
$ mkdir ~/bin
$ PATH=~/bin:$PATH
```

下载 repo 的脚本，用于执行 repo，命令如下：

```
$ curl https://dl-ssl.google.com/dl/googlesource/git-repo/repo > ~/bin/repo
```

设置可执行权限，命令如下：

```
$ chmod a+x ~/bin/repo
```

（2）初始化一个 repo 的客户端

在用户目录下，创建一个空目录，用于存放 Android 源码，命令如下：

```
$ mkdir AndroidCode
$ cd AndroidCode
```

进入到 AndroidCode 目录，并运行 repo 下载源码，下载主线分支的代码，主线分支包括最新修改的 bug，以及并未正式发布版本的最新源码，命令如下：

```
$ repo init -u https://android.googlesource.com/platform/manifest
```

下载其他分支，正式发布的版本可以通过添加-b 参数来下载，命令如下：

```
$ repo init -u https://android.googlesource.com/platform/manifest -b
                                              android-5.0.0_r1
```

例如，可以使用如下命令来初始化最新 Android 源代码。

```
./repo init -u https://android.googlesource.com/platform/manifest -b android-5.0_r1
```

输入上面的命令后按 Enter 键执行，如图 1-1 所示。

```
guan@guan-PC /cygdrive/g/android/bin/aa
$ repo init -u https://android.googlesource.com/platform/manifest -b android-5.0.0_r1
warning: gpg (GnuPG) is not available.
warning: Installing it is strongly encouraged.

Get https://gerrit.googlesource.com/git-repo
remote: Counting objects: 117, done
remote: Finding sources: 100% (117/117)
remote: Total 2955 (delta 1549), reused 2955 (delta 1549)
接收对象中: 100% (2955/2955), 2.51 MiB | 33.00 KiB/s, 完成.
```

图 1-1　选择下载的分支

在下载过程中需要填写 Name 和 Email，填写完毕之后，选择 Y 进行确认，最后提示 repo 初始化完成，这时可以开始同步 Android 源码了，同步过程较耗费时间，需要耐心等待，执行下面命令开始同步代码。

```
$ repo sync
```

经过上述步骤后，便开始下载并同步 Android 源码，界面效果如图 1-2 所示。

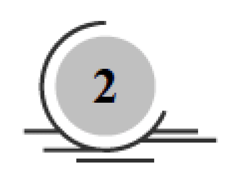
```
/cygdrive/g/android/bin/aa
 * [新tag]          android-cts-4.2_r1 -> android-cts-4.2_r1
 * [新tag]          android-cts-4.2_r2 -> android-cts-4.2_r2
 * [新tag]          android-cts-4.4_r1 -> android-cts-4.4_r1
 * [新tag]          android-cts-verifier-4.0.3_r1 -> android-cts-verifier-4.0.3_r1
 * [新tag]          android-cts-verifier-4.0_r1 -> android-cts-verifier-4.0_r1
 * [新tag]          android-l-preview_r2 -> android-l-preview_r2
 * [新tag]          android-sdk-4.0.3-tools_r1 -> android-sdk-4.0.3-tools_r1
 * [新tag]          android-sdk-4.0.3_r1 -> android-sdk-4.0.3_r1
 * [新tag]          android-sdk-4.4.2_r1 -> android-sdk-4.4.2_r1
 * [新tag]          android-sdk-4.4.2_r1.0.1 -> android-sdk-4.4.2_r1.0.1
 * [新tag]          android-sdk-adt_r16.0.1 -> android-sdk-adt_r16.0.1
 * [新tag]          android-sdk-adt_r20 -> android-sdk-adt_r20
 * [新tag]          android-sdk-support_r11 -> android-sdk-support_r11
Fetching projects:   4% (20/482)  Fetching project platform/dalvik
 25  328M   25 82.2M    0     0   100k      0  0:55:39  0:13:54  0:41:45 75486curl: (35) Unknow
n SSL protocol error in connection to android.googlesource.com:443
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100     1    0     1    0     0      0      0 --:--:--  0:00:01 --:--:--     0
  0     0    0     0    0     0      0      0 --:--:--  0:00:05 --:--:--     0remote: Sending a
pproximately 114.87 MiB ...
remote: Counting objects: 4376, done
remote: Finding sources: 100% (84/84)
  0     0    0     0    0     0      0      0 --:--:--  0:00:06 --:--:--     0接收对象中:   0%
  0     0    0     0    0     0      0      0 --:--:--  0:00:07 --:--:--     0接收对象中:   0%
 25  328M   25 83.9M    0     0   101k      0  0:55:11  0:14:04  0:41:07  192k接收对象中:   0%
  0     0    0     0    0     0      0      0 --:--:--  0:00:10 --:--:--     0接收对象中:   0%
  0     0    0     0    0     0      0      0 --:--:--  0:00:11 --:--:--     0接收对象中:   0%
  0     0    0     0    0     0      0      0 --:--:--  0:00:12 --:--:--     0接收对象中:   0%
  0     0    0     0    0     0      0      0 --:--:--  0:00:13 --:--:--     0接收对象中:   0%
 25  328M   25 84.8M    0     0   102k      0  0:55:00  0:14:10  0:40:50  157k接收对象中:   0%
  0     0    0     0    0     0      0      0 --:--:--  0:00:15 --:--:--     0接收对象中:   0%
  0     0    0     0    0     0      0      0 --:--:--  0:00:16 --:--:--     0接收对象中:   0%
  0     0    0     0    0     0      0      0 --:--:--  0:00:17 --:--:--     0接收对象中:   0%
  0     0    0     0    0     0      0      0 --:--:--  0:00:18 --:--:--     0接收对象中:   0%
  0     0    0     0    0     0      0      0 --:--:--  0:00:19 --:--:--     0接收对象中:   1%
 25  328M   25 85.3M    0     0   101k      0  0:55:03  0:14:16  0:40:47 76923接收对象中:   1%
  0     0    0     0    0     0      0      0 --:--:--  0:00:21 --:--:--     0接收对象中:   1%
 26  328M   26 85.5M    0     0   102k      0  0:55:01  0:14:18  0:40:43 88897接收对象中:   1%
  0     0    0     0    0     0      0      0 --:--:--  0:00:23 --:--:--     0接收对象中:   1%
  0     0    0     0    0     0      0      0 --:--:--  0:00:24 --:--:--     0接收对象中:   1% (8150/462517)
  0     0    0     0    0     0      0      0 --:--:--  0:00:25 --:--:--     0接收对象中:   1% (8855/462517)
  0     0    0     0    0     0      0      0 --:--:--  0:00:26 --:--:--     0接收对象中:   2% (9251/462517)
 26  328M   26 86.1M    0     0   102k      0  0:54:56  0:14:23  0:40:33  139k
```

图 1-2　正在下载源码

因为网络方面的原因，可能执行./repo init -u https://android.googlesource.com/platform/manifest -b android-5.0.0_r1 初始化命令会失败，提示一些类似网络连接失败的信息，此时不用理会，只需继续执行这个命令。如果出现多次失败提示，则可以尝试使用如下方法解决。

（1）使用如下命令删除 Android 5.0 文件中的缓存文件，然后重新执行初始化命令。

```
rm -rf * -R
```



（2）隔一段时间或者晚上、凌晨时下载，一般这个时段更容易下载 Android 源代码。

如果看到类似如图 1-3 所示的信息，则表示连接成功，正在初始化。

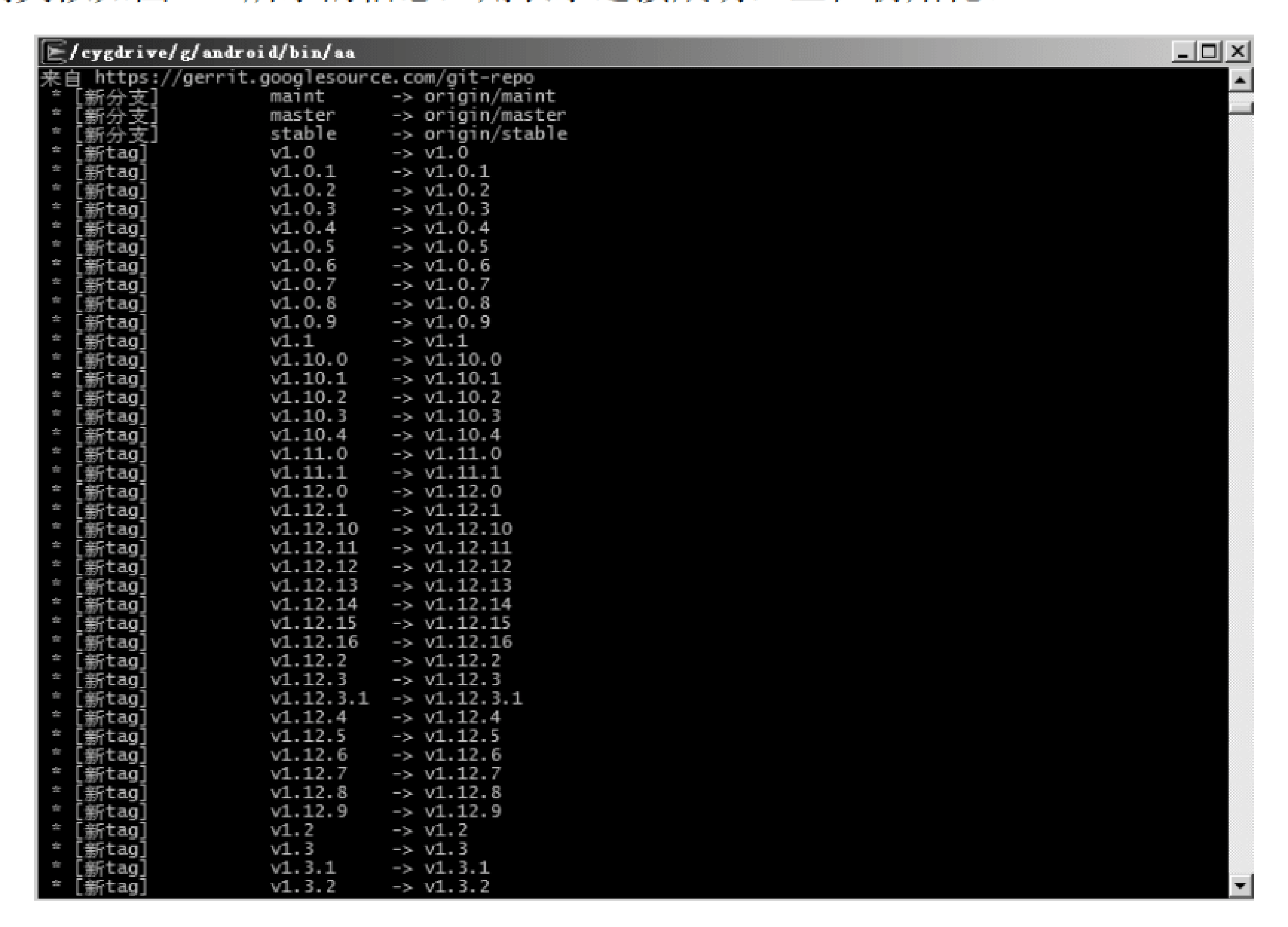


图 1-3 成功初始化

注意：

（1）在源代码下载过程中，源代码下载目录中看不到任何文件，打开“显示/隐藏”，会看到一个名为.repo 的文件夹，这个文件夹用来保存 Android 源代码的“临时文件”。

（2）当文件最后下载接近完成时，会从.repo 文件夹中导出 Android 源代码。

（3）当下载 Android 5.0 源代码完成后，可以看到 Android 源代码下载目录中会有 bionic、bootable、build、cts、dalvik 等文件夹目录，这些就是 Android 的源代码。

（4）如果不得不关闭计算机停止下载，那么可以在源代码下载的终端按下 Ctrl+C 或者 Ctrl+Z 快捷键停止源代码的下载，这样不会造成源代码的丢失或损坏。

1.1.2 在 Windows 平台获取 Android 源码

在 Windows 平台上获取 Android 源码的方式和在 Linux 中的获取原理相同，但是需要预先在 Windows 平台上面搭建一个 Linux 环境，此处需要用到 cygwin 工具。cygwin 的作用是构建一套在 Windows 上的 Linux 模拟环境，下载 cygwin 工具的地址是 http://cygwin.com/install.html。

下载成功后会得到一个名为 setup.exe 的可执行文件，经过此文件可以更新和下载最新的工具版本，具体流程如下所示。

（1）启动 cygwin，如图 1-4 所示。

（2）单击“下一步”按钮，选择第一个选项：从网络下载安装，如图 1-5 所示。

（3）单击“下一步”按钮，选择安装根目录，如图 1-6 所示。

（4）单击“下一步”按钮，选择临时文件目录，如图 1-7 所示。



图 1-4　启动 cygwin

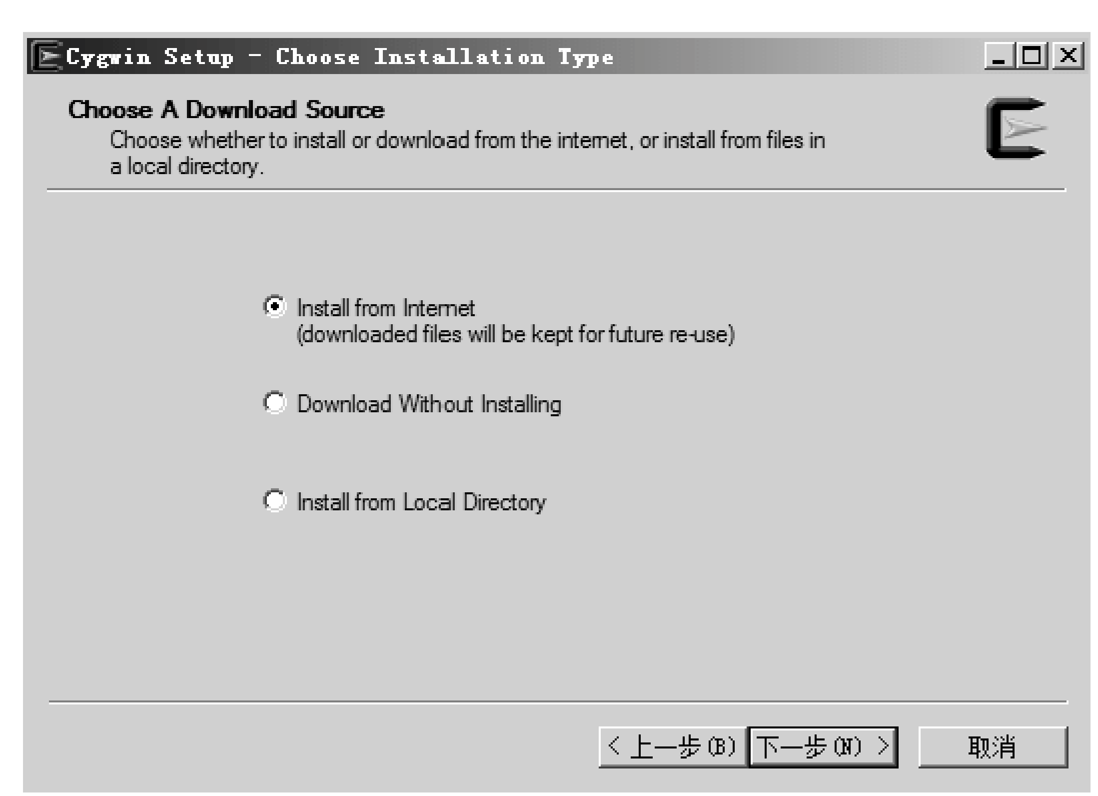


图 1-5　选择从网络下载安装

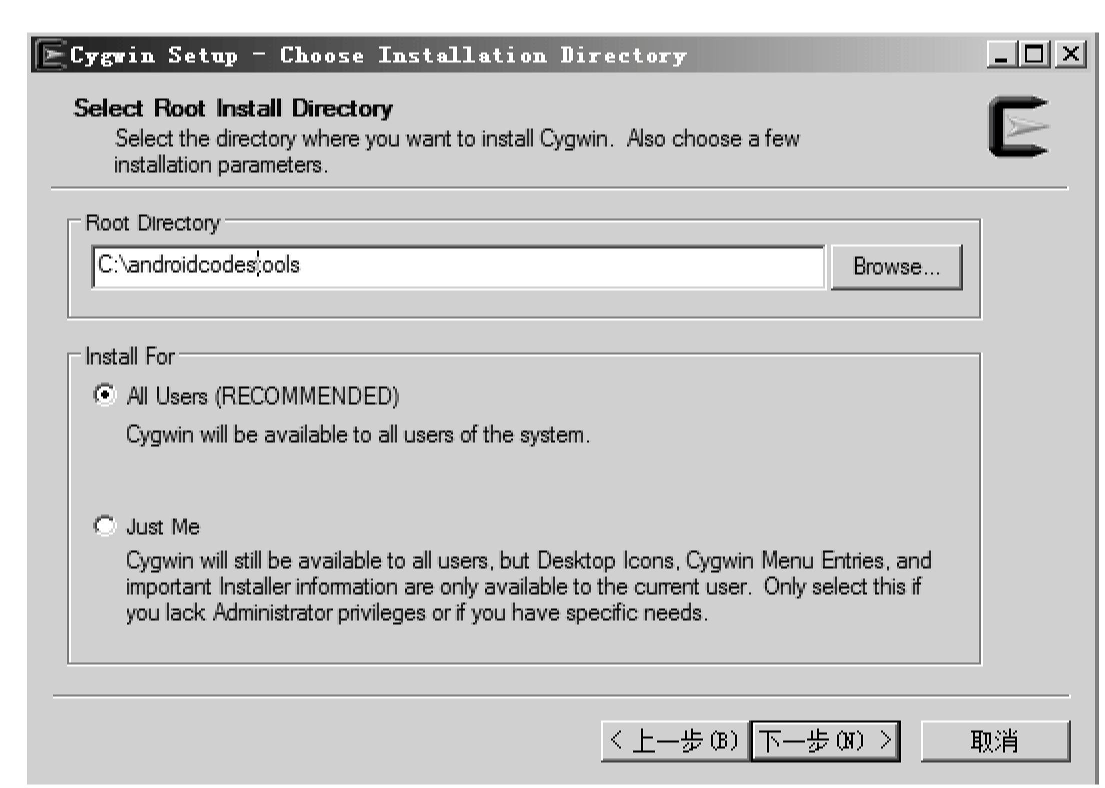


图 1-6　选择安装根目录

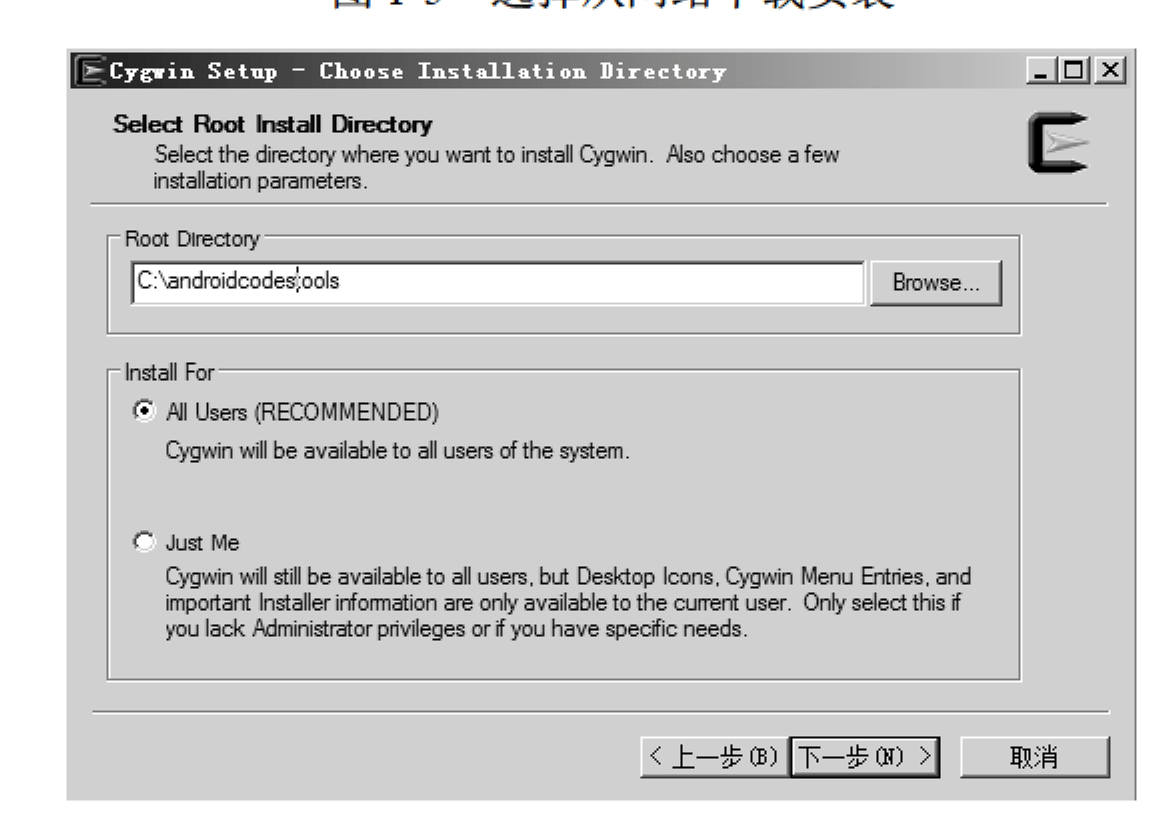


图 1-7　选择临时文件目录

（5）单击“下一步”按钮，设置网络代理。如果所在网络需要代理，则在这一步进行设置，如果不用代理，则选择直接下载，如图 1-8 所示。

（6）单击“下一步”按钮，选择下载站点。一般选择比较近的站点速度会比较快，这里选择的是台湾站点，如图 1-9 所示。

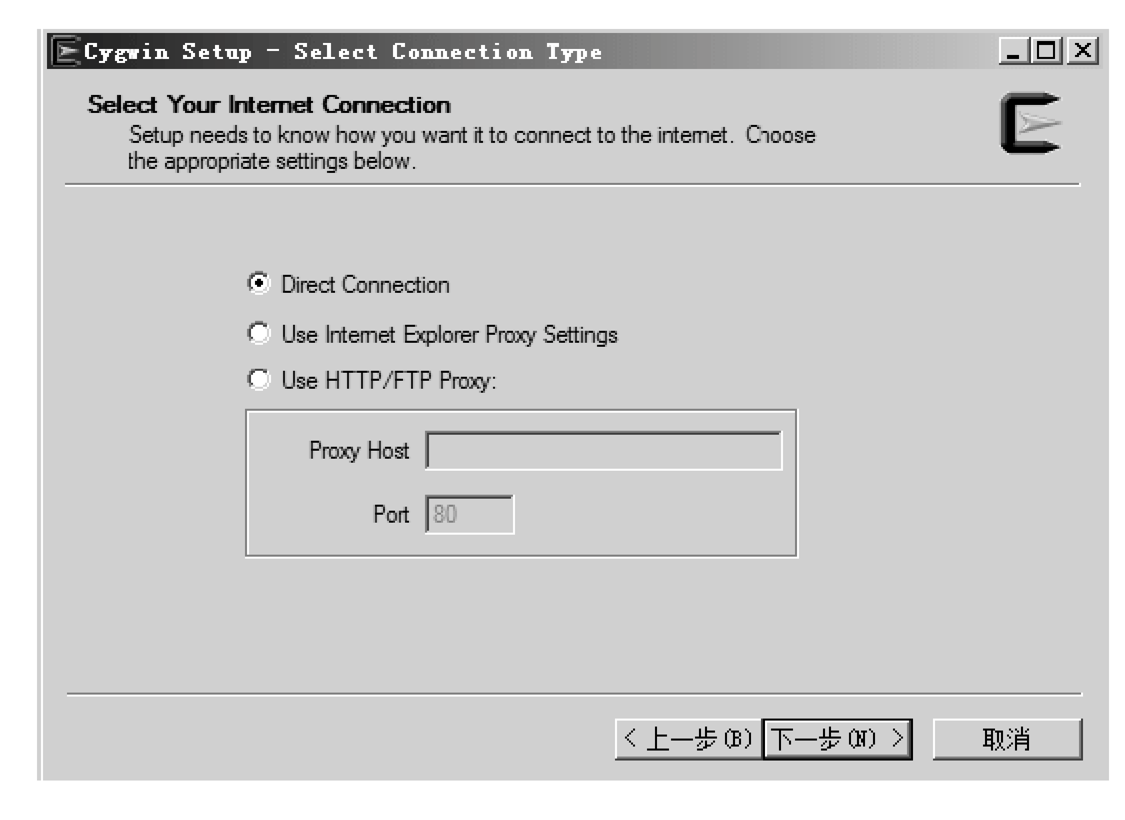


图 1-8　设置网络代理

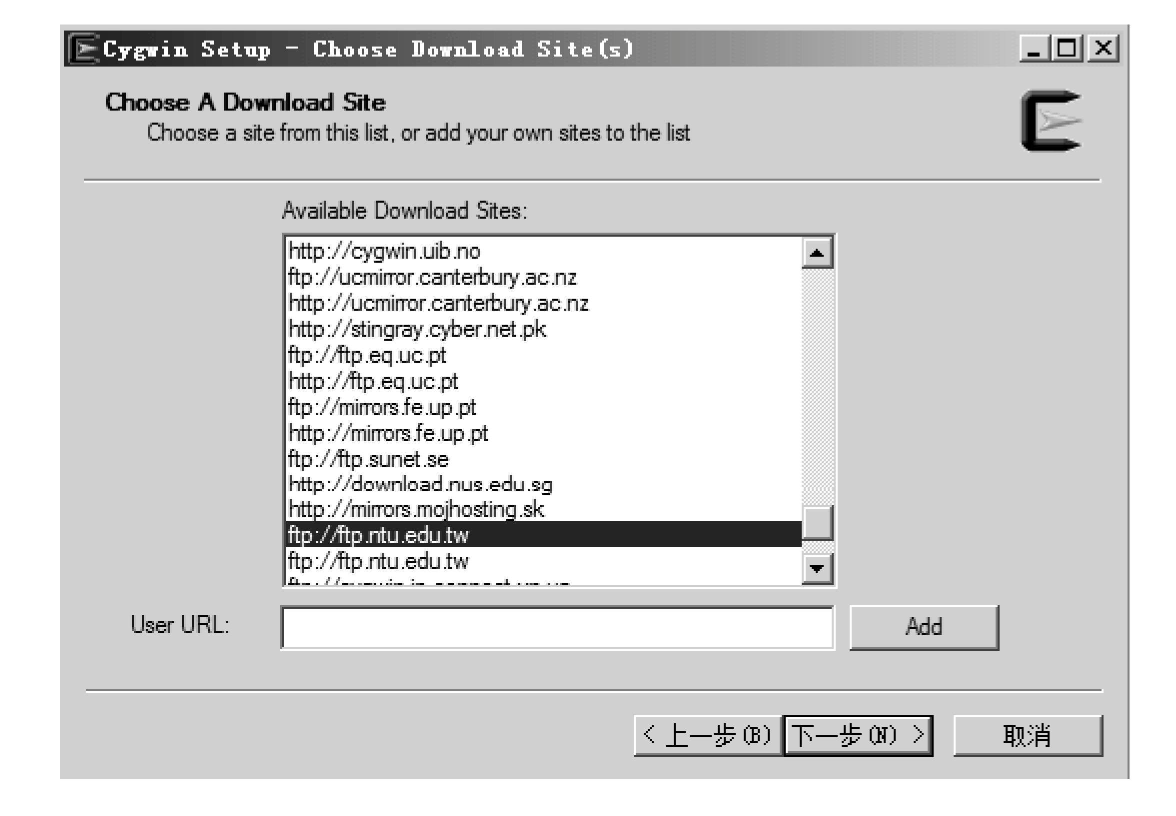
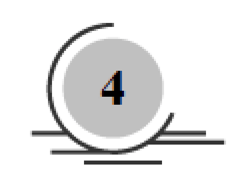


图 1-9　选择下载站点

（7）单击“下一步”按钮，开始更新工具列表，如图 1-10 所示。



（8）单击“下一步”按钮，选择需要下载的工具包。在此需要依次下载 curl、git、python 这些工具，如图 1-11 所示。

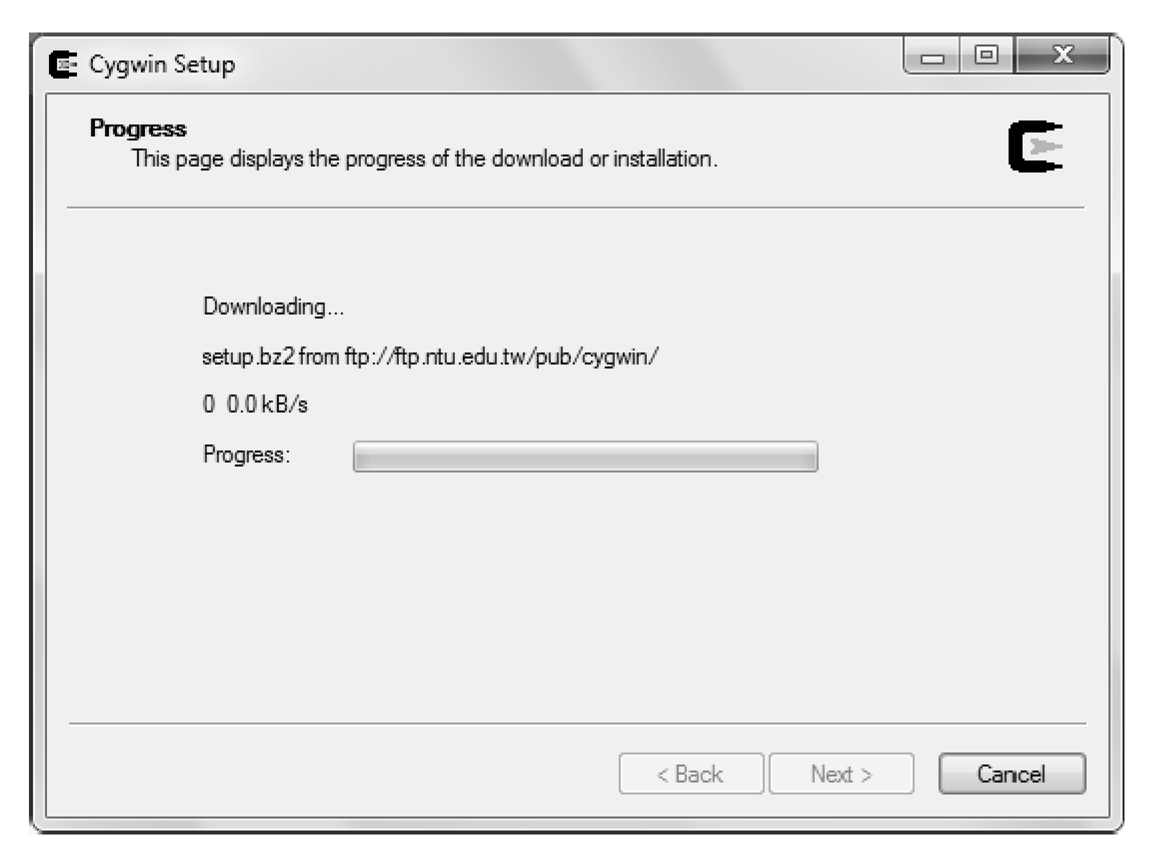


图 1-10　更新工具列表

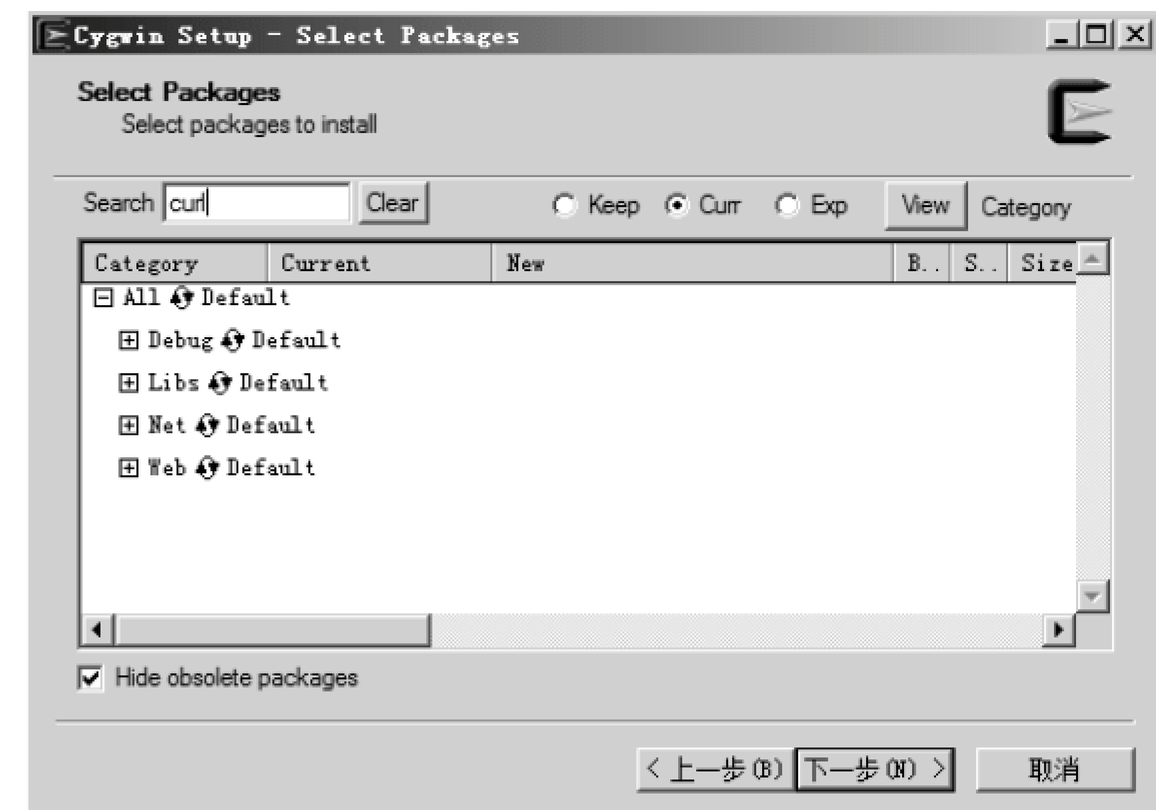


图 1-11　依次下载工具

为了确保能够安装上述工具，一定要用鼠标双击变为 Install 形式，如图 1-12 所示。

（9）单击“下一步”按钮，开始下载安装，如图 1-13 所示。

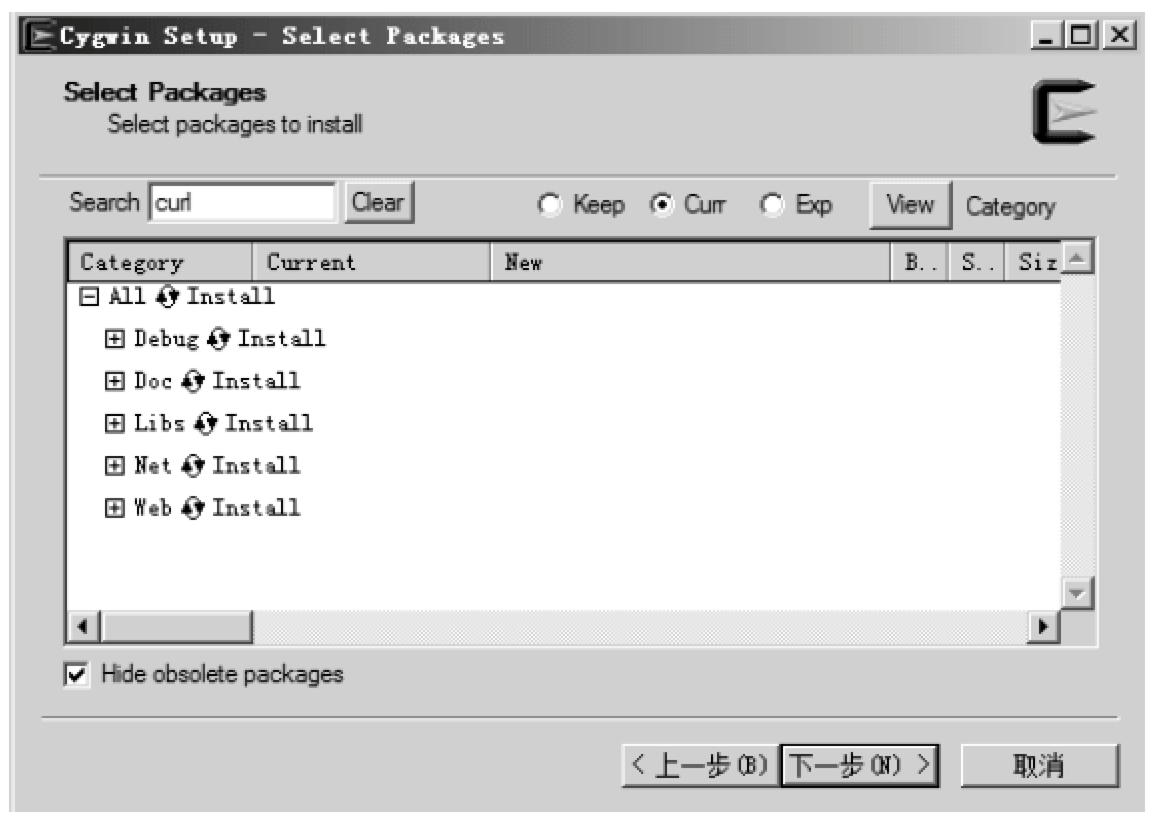


图 1-12　设置为 Install 形式

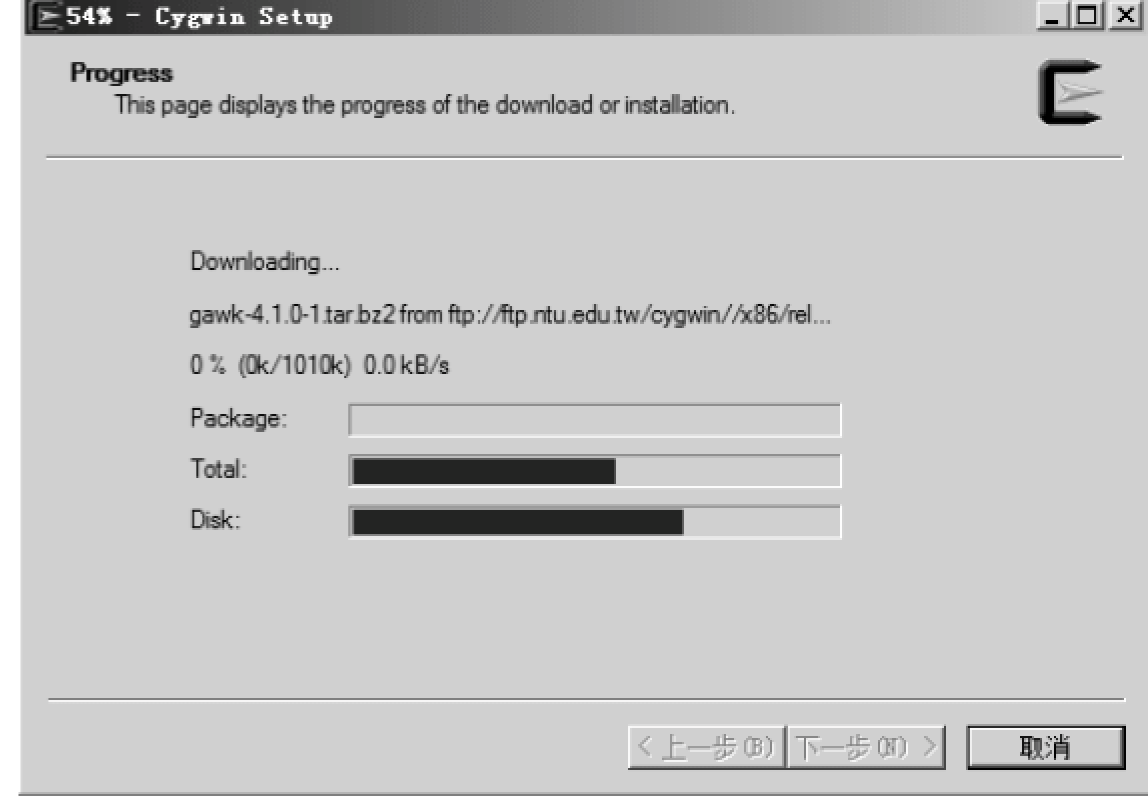


图 1-13　下载进度条

下载安装成功会出现提示信息，单击“完成”按钮即完成安装。当安装好 cygwin 后，打开 cygwin，会模拟出一个 Linux 的工作环境，然后按照 Linux 平台的源码下载方法即可下载 Android 源码。

建议读者在下载 Android 源码时，严格按照官方提供的步骤进行，地址是 http://source.android.com/source/downloading.html，这一点对初学者来说尤为重要。另外，整个下载过程比较漫长，需要大家耐心等待。如图 1-14 所示是笔者机器中的命令截图。

```
Cloning into 'deb'...
remote: Counting objects: 22, done
remote: Finding sources: 100% (22/22)
remote: Total 1018 (delta 416), reused 1018 (delta 416)R
Receiving objects:  99% (1008/1018)
Receiving objects: 100% (1018/1018), 272.73 KiB | 0 bytes/s, done.
Resolving deltas: 100% (416/416), done.
Checking connectivity... done.
Cloning into 'flo'...
remote: Counting objects: 22, done
remote: Finding sources: 100% (22/22)
rRemote: Total 2827 (delta 1340), reused 2827 (delta 1340)eceiving objects:  99%
Receiving objec07.79 KiB | 1.30 MiB/s
Receiving objects: 100% (2827/2827), 1.22 MiB | 1.30 MiB/s, done.
Resolving deltas: 100% (1340/1340), done.
Checking connectivity... done.
Cloning into 'flo-kernel'...
remote: Sending approximately 984.72 MiB ...
remote: Counting objects: 22, done
remote: Finding sources: 100% (22/22)
Receiving objects:  97% (802/820), 879.35 MiB | 1.39 MiB/s
```

图 1-14　在 Windows 中用 cygwin 工具下载 Android 源码

1.2　分析 Android 源码结构

获得 Android 5.0 源码后，可将源码的全部工程分为如下 3 个部分。

- ☑ Core Project：核心工程部分，这是建立 Android 系统的基础，被保存在根目录的各个文件夹中。
- ☑ External Project：扩展工程部分，可以使其他开源项目具有扩展功能，被保存在 external 文件夹中。
- ☑ Package：包部分，提供了 Android 的应用程序、内容提供者、输入法和服务，被保存在 package 文件夹中。

本节将详细讲解 Android 5.0 源码的目录结构。

1.2.1　总体结构

无论是 Android 1.5 还是 Android 5.0，各个版本的源码目录基本类似。在里面包含了原始 Android 的目标机代码、主机编译工具和仿真环境。解压缩下载的 Android 5.0 源码包后，可以看到在第一级目录中有多个文件夹和一个 Makefile 文件，如图 1-15 所示。

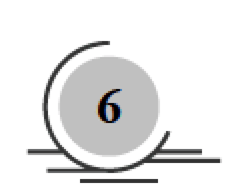


图 1-15　下载的 Android 5.0 源码

如果是编译后的源码目录，则会增加一个 out 文件夹，用来存放编译产生的文件。Android 5.0 第一级别目录结构的具体说明如表 1-1 所示。

表 1-1　Android 5.0 源码的根目录

Android 源码根目录	描　述
abi	abi 相关代码，abi:application binary interface，应用程序二进制接口
art	全新的运行环境，需要和 Dalvik VM 区分开来
bionic	bionic C 库
bootable	启动引导相关代码
build	存放系统编译规则及 generic 等基础开发配置包
cts	Android 兼容性测试套件标准
dalvik	dalvik Java 虚拟机
development	应用程序开发相关
device	设备相关代码
docs	介绍开源的相关文档
external	Android 使用的一些开源的模组
frameworks	核心框架——Java 及 C++语言，是 Android 应用程序的框架
gdk	即时通信模块
hardware	主要是硬件适配层 HAL 代码



续表

Android 源码根目录	描　　述
kernel	Linux 的内核文件
libcore	核心库相关
libnativehelper	是 Support functions for Android's class libraries 的缩写，表示动态库，是实现的 JNI 库的基础
ndk	ndk 相关代码。Android NDK（Android Native Development Kit）是一系列的开发工具，允许程序开发人员在 Android 应用程序中嵌入 C/C++语言编写的非托管代码
out	编译完成后的代码输出在此目录
packages	应用程序包
pdk	Plug Development Kit 的缩写，是本地开发套件
prebuilts	x86 和 arm 架构下预编译的一些资源
sdk	sdk 及模拟器
system	文件系统和应用及组件，是用 C 语言实现的
tools	工具文件夹
vendor	厂商定制代码
Makefile	全局的 Makefile

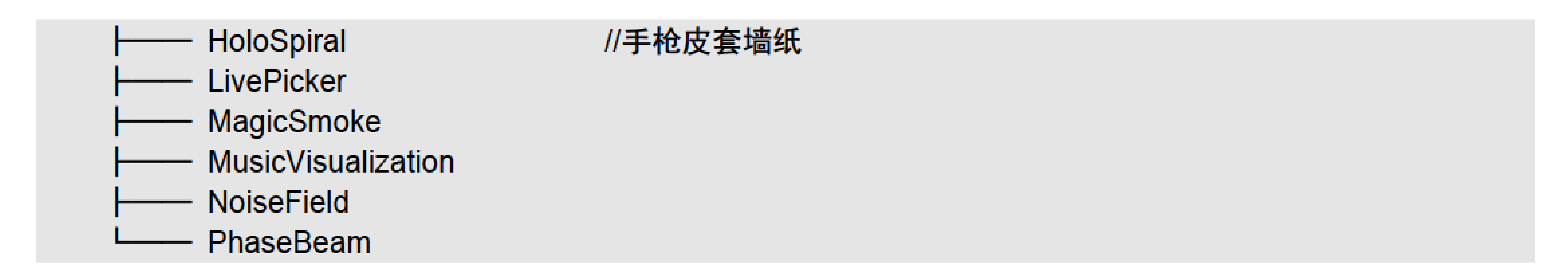
由此可见，通过对源码中根目录的每个文件夹功能的介绍，可以看出源码按功能分类还是非常清晰的，可以分为系统代码、工具、文档、开发环境、虚拟机、配置脚本和编译脚本等类别，并且也可以看出涉及的内容比较庞大和复杂，源码分析工作需要多方面的理论和实践知识。

1.2.2　应用程序部分

应用程序主要是 UI 界面的实现，广大开发者基于 SDK 上开发的一个个独立的 APK 包，都是属于应用程序这一层的，应用程序在 Android 系统中处于最上层的位置。源码结构中的 packages 目录用来实现系统的应用程序，packages 的目录结构如下所示。

```
packages /
├─── apps //应用程序库
│    ├─── BasicSmsReceiver        //基础短信接收
│    ├─── Bluetooth               //蓝牙
│    ├─── Browser                 //浏览器
│    ├─── Calculator              //计算器
│    ├─── Calendar                //日历
│    ├─── Camera                  //照相机
│    ├─── CellBroadcastReceiver   //单元广播接收
│    ├─── CertInstaller           //被调用的包，在 Android 中安装数字签名
│    ├─── Contacts                //联系人
│    ├─── DeskClock               //桌面时钟
│    ├─── Email                   //电子邮件
│    ├─── Exchange                //Exchange 服务
│    ├─── Gallery                 //图库
│    ├─── Gallery2                //图库 2
│    ├─── HTMLViewer              //HTML 查看器
│    ├─── KeyChain                //密码管理
```

```
|       ├──── Launcher2                   //启动器 2
|       ├──── Mms                         //彩信
|       ├──── Music                       //音乐
|       ├──── MusicFX                     //音频增强
|       ├──── Nfc                         //近场通信
|       ├──── PackageInstaller            //包安装器
|       ├──── Phone                       //电话
|       ├──── Protips                     //主屏幕提示
|       ├──── Provision                   //引导设置
|       ├──── QuickSearchBox              //快速搜索框
|       ├──── Settings                    //设置
|       ├──── SoundRecorder               //录音机
|       ├──── SpareParts                  //系统设置
|       ├──── SpeechRecorder              //录音程序
|       ├──── Stk                         //SIM 卡相关
|       ├──── Tag                         //标签
|       ├──── VideoEditor                 //视频编辑
|       └──── VoiceDialer                 //语音编号
├──── experimental                        //非官方的应用程序
|       ├──── BugReportSender             //bug 的报告程序
|       ├──── Bummer
|       ├──── CameraPreviewTest           //照相机预览测试程序
|       ├──── DreamTheater
|       ├──── ExampleImsFramework
|       ├──── LoaderApp
|       ├──── NotificationLog
|       ├──── NotificationShowcase
|       ├──── procstatlog
|       ├──── RpcPerformance
|       └──── StrictModeTest
├──── inputmethods                        //输入法
|       ├──── LatinIME                    //拉丁文输入法
|       ├──── OpenWnn                     //OpenWnn 输入法
|       └──── PinyinIME                   //拼音输入法
├──── providers                           //提供器
|       ├──── ApplicationsProvider        //应用程序提供器，提供应用程序所需的界面
|       ├──── CalendarProvider            //日历提供器
|       ├──── ContactsProvider            //联系人提供器
|       ├──── DownloadProvider            //下载管理提供器
|       ├──── DrmProvider                 //数据库相关
|       ├──── GoogleContactsProvider      //Google 联系人提供器
|       ├──── MediaProvider               //媒体提供器
|       ├──── TelephonyProvider           //彩信提供器
|       └──── UserDictionaryProvider      //用户字典提供器
├──── screensavers                        //屏幕保护
|       ├──── Basic                       //基本屏幕保护
|       ├──── PhotoTable                  //照片方格
|       ├──── WebView                     //网页
└──── wallpapers                          //墙纸
      ├──── Basic                         //系统内置墙纸
      ├──── Galaxy4                       //S4 内置墙纸
```



```
    ├──── HoloSpiral                    //手枪皮套墙纸
    ├──── LivePicker
    ├──── MagicSmoke
    ├──── MusicVisualization
    ├──── NoiseField
    └──── PhaseBeam
```

通过上面的目录结构可以看出，package 目录主要存放的是 Android 系统应用层相关的内容，包括应用程序相关的包或者资源文件，其中既包括系统自带的应用程序，又有第三方开发的应用程序，还有屏幕保护和墙纸等应用，所以源码中 package 目录对应着系统的应用层。

1.2.3　应用程序框架部分

应用程序框架是 Android 系统中的核心部分，也就是 SDK 部分，会提供接口给应用程序使用，同时应用程序框架又会和系统服务、系统程序库、硬件抽象层有关联，所以其作用十分重要，应用程序框架的实现代码大部分都在/frameworks/base 和/frameworks/av 目录下，/frameworks/base 的目录结构如下所示。

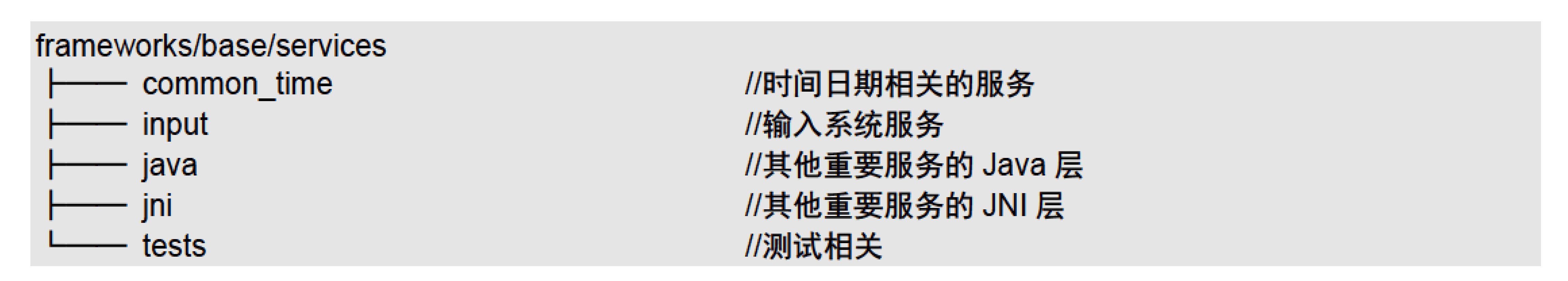
```
frameworks/base
├──── api                   //全是 XML 文件，定义了 API
├──── cmds                  //Android 中的重要命令（am、app_proce 等）
├──── core                  //核心库
├──── data                  //声音字体等数据文件
├──── docs                  //文档
├──── drm                   //数字版权管理
├──── graphics              //图形图像
├──── icu4j                 //用于解决国际化问题
├──── include               //头文件
├──── keystore              //数字签名证书相关
├──── libs                  //库
├──── location              //地理位置
├──── media                 //多媒体
├──── native                //本地库
├──── nfc-extras            //NFC 相关
├──── obex                  //蓝牙传输
├──── opengl                //OpenGL 相关
├──── packages              //设置，TTS，VPN 程序
├──── policy                //锁屏界面相关
├──── sax                   //XML 解析器
├──── services              //Android 的服务
├──── telephony             //电话相关
├──── test-runner           //测试相关
├──── tests                 //测试相关
├──── tools                 //工具
├──── voip                  //可视通话
└──── wifi                  //无线网络
```

以上文件夹包含了应用程序框架层的大部分代码，正是这些目录下的文件构成了 Android 的应用

程序框架层，暴露出接口给应用程序调用，同时衔接系统程序库和硬件抽象层，形成一个由上至下的调用过程。在/frameworks/base 目录下也涉及系统服务，程序库中的一些代码将在后面的两个小节中详细分析。

1.2.4 系统服务部分

在 1.2.3 中介绍了应用程序框架层的内容，了解到大部分的实现代码保存在/frameworks/base 目录下。其实在这个目录中还有一个名为 service 的目录，其中的代码是用于实现 Android 系统服务的。接下来将详细介绍 service 目录下的内容，其目录结构如下所示。



```
frameworks/base/services
├── common_time                              //时间日期相关的服务
├── input                                    //输入系统服务
├── java                                     //其他重要服务的 Java 层
├── jni                                      //其他重要服务的 JNI 层
└── tests                                    //测试相关
```

其中，java 和 jni 两个目录分别是一些其他服务的 Java 层和 JNI 层实现，java 目录下更详细的目录结构以及其他 Android 系统服务的说明如下所示。

```
frameworks/base/services/java/com/android/server
├── accessibility
├── am
├── connectivity
├── display
├── dreams
├── drm
├── input
├── location
├── net
├── pm
├── power
├── updates
├── usb
├── wm
├── AlarmManagerService.java                 //闹钟服务
├── AppWidgetService.java                    //应用程序小工具服务
├── AppWidgetServiceImpl.java
├── AttributeCache.java
├── BackupManagerService.java                //备份服务
├── BatteryService.java                      //电池相关服务
├── BluetoothManagerService.java             //蓝牙
├── BootReceiver.java
├── BrickReceiver.java
├── CertBlacklister.java
├── ClipboardService.java
├── CommonTimeManagementService.java         //时间管理服务
├── ConnectivityService.java
├── CountryDetectorService.java
```

```
├──── DevicePolicyManagerService.java
├──── DeviceStorageMonitorService.java        //设备存储器监听服务
├──── DiskStatsService.java                   //磁盘状态服务
├──── DockObserver.java                       //底座监视服务
├──── DropBoxManagerService.java
├──── EntropyMixer.java
├──── EventLogTags.logtags
├──── INativeDaemonConnectorCallbacks.java
├──── InputMethodManagerService.java          //输入法管理服务
├──── IntentResolver.java
├──── IntentResolverOld.java
├──── LightsService.java
├──── LocationManagerService.java             //地理位置服务
├──── MasterClearReceiver.java
├──── MountService.java                       //挂载服务
├──── NativeDaemonConnector.java
├──── NativeDaemonConnectorException.java
├──── NativeDaemonEvent.java
├──── NetworkManagementService.java           //网络管理服务
├──── NetworkTimeUpdateService.java
├──── NotificationManagerService.java         //通知服务
├──── NsdService.java
├──── PackageManagerBackupAgent.java
├──── PreferredComponent.java
├──── ProcessMap.java
├──── RandomBlock.java
├──── RecognitionManagerService.java
├──── SamplingProfilerService.java
├──── SerialService.java                      //NFC 相关
├──── ServiceWatcher.java
├──── ShutdownActivity.java
├──── StatusBarManagerService.java            //状态栏管理服务
├──── SystemBackupAgent.java
├──── SystemServer.java
├──── TelephonyRegistry.java
├──── TextServicesManagerService.java
├──── ThrottleService.java
├──── TwilightCalculator.java
├──── TwilightService.java
├──── UiModeManagerService.java
├──── UpdateLockService.java                  //锁屏更新服务
├──── VibratorService.java                    //振动服务
├──── WallpaperManagerService.java            //壁纸服务
├──── Watchdog.java                           //看门狗
├──── WifiService.java                        //无线网络服务
└──── WiredAccessoryManager.java              //无线设备管理服务
```

从上面的文件夹和文件可以看出，Android 中涉及的服务种类非常多，包括界面、网络、电话等核心模块基本上都有其专属的服务，这些是属于系统级别的服务，这些系统服务一般都会在 Android 系统启动时加载，在系统关闭时结束，受到系统的管理，应用程序并没有权力去打开或者关闭，它们会

随着系统的运行一直在后台运行，供应用程序和其他组件来使用。

另外，在 frameworks/av/下面也有一个 services 目录，这个目录下存放的是音频和照相机的服务的实现代码，目录结构如下所示。

```
frameworks/av/services
├── audioflinger                    //音频管理服务
└── camera                          //照相机的管理服务
```

这个 av/services 目录下的文件主要是用来支持 Android 系统中的音频和照相机服务的，这是两个非常重要的系统服务，开发应用程序时会经常依赖这两个服务。

1.2.5 系统程序库部分

Android 的系统程序库类型非常多，功能也非常强大，正是有了这些程序库，Android 系统才能运行多种多样的应用程序。下面笔者挑了一些很常用也很重要的系统程序库来分析它们在源码中所处的位置。

（1）系统 C 库

Android 系统采用的是一个从 BSD 继承而来的标准的系统函数库 bionic，在源码根目录下有这个文件夹，其目录结构如下所示。

```
bionic/
├── libc                            //C 库
├── libdl                           //动态链接库相关
├── libm                            //数学库
├── libstdc++                       //C++实现库
├── libthread_db                    //线程库
├── linker                          //连接器相关
└── test                            //测试相关
```

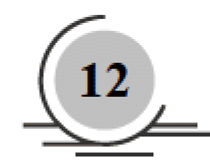
（2）媒体库

Android 中的媒体库在 2.3 版本之前是由 OpenCore 实现的，2.3 版本之后 Stragefright 被替换了，OpenCore 成为了新的多媒体的实现库。同时 Android 也自带了一些音频/视频的管理库，用于管理多媒体的录制、播放、编码和解码等功能。Android 的多媒体程序库的实现代码主要在/frameworks/av/media 目录下，其目录结构如下所示。

```
frameworks/av/media/
├── common_time                     //时间相关
├── libeffects                      //多媒体效果
├── libmedia                        //多媒体录制，播放
├── libmedia_native                 //里面只有一个 Android.mk，用来编译 native 文件
├── libmediaplayerservice           //多媒体播放服务的实现库
├── libstagefright                  //stagefright 的实现库
├── mediaserver                     //跨进程多媒体服务
└── mtp                             //mtp 协议的实现（媒体传输协议）
```

（3）图层显示库

Android 中的图层显示库主要负责对显示子系统的管理，负责图层的渲染、叠加、绘制等功能，提



供了 2D 和 3D 图层的无缝融合，是整个 Android 系统显示的“大脑中枢”，其代码在/frameworks/native/services/surfaceflinger/目录下，其目录结构如下所示。

```
frameworks/native/services/surfaceflinger/
├── DisplayHardware                   //显示底层相关
├── tests                             //测试
├── Android.mk                        //MakeFile 文件
├── Barrier.h
├── Client.cpp                        //显示的客户端实现文件
├── Client.h
├── clz.cpp
├── clz.h
├── DdmConnection.cpp
├── DdmConnection.h
├── DisplayDevice.cpp                 //显示设备相关
├── DisplayDevice.h
├── EventThread.cpp                   //消息线程
├── EventThread.h
├── GLExtensions.cpp                  //OpenGL 扩展
├── GLExtensions.h
├── Layer.cpp                         //图层相关
├── Layer.h
├── LayerBase.cpp                     //图层基类
├── LayerBase.h
├── LayerDim.cpp                      //图层相关
├── LayerDim.h
├── LayerScreenshot.cpp               //图层相关
├── LayerScreenshot.h
├── MessageQueue.cpp                  //消息队列
├── GLExtensions.h
├── MessageQueue.h
├── MODULE_LICENSE_APACHE2            //证书
├── SurfaceFlinger.cpp                //图层管理者，图层管理的核心类
├── SurfaceFlinger.h
├── SurfaceTextureLayer.cpp           //文字图层
├── SurfaceTextureLayer.h
├── Transform.cpp
└── Transform.h
```

（4）网络引擎库

网络引擎库主要是用来实现 Web 浏览器的引擎，支持 Android 的 Web 浏览器和一个可嵌入的 Web 视图，这个是采用第三方开发的浏览器引擎 Webkit 实现的，Webkit 的代码在/external/webkit/目录下，其目录结构如下所示。

```
external/webkit/
├── Examples                          //Webkit 示例
├── LayoutTests                       //布局测试
├── PerformanceTests                  //表现测试
├── Source                            //Webkit 源代码
├── Tools                             //工具
```

```
├── WebKitLibraries                       //Webkit 用到的库
├── Android.mk                            //Makefile
├── bison_check.mk
├── CleanSpec.mk
├── MODULE_LICENSE_LGPL                   //证书
├── NOTICE
└── WEBKIT_MERGE_REVISION                 //版本信息
```

（5）3D 图形库

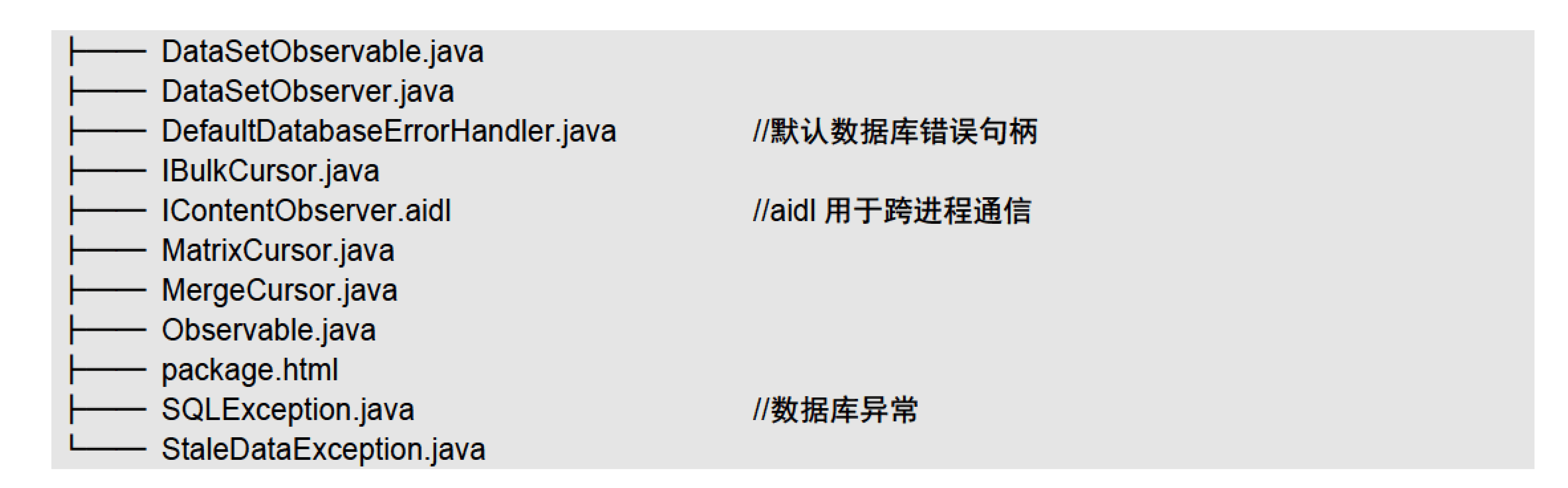
Android 中的 3D 图形渲染是采用 OpenGL 来实现的，OpenGL 是开源的第三方图形渲染库，使用该库可以实现 Android 中的 3D 图形硬件加速或者 3D 图形软件加速功能，是一个非常重要的功能库。从 Android 5.0 开始，支持最新最强大的 OpenGL ES 3.1。其实现代码在/frameworks/native/opengl 中，其目录结构如下所示。

```
frameworks/native/opengl/
├── include                               //OpenGL 中的头文件
├── libagl                                //在 MAC os 上的库
├── libs                                  //OpenGL 的接口和实现库
├── specs                                 //OpenGL 的文档
├── tests                                 //测试相关
└── tools                                 //工具库
```

（6）SQLite

SQLite 是 Android 系统自带的一个轻量级关系数据库，其实现源代码已经在网上开源。SQLite 的优点是操作简单方便，运行速度较快，占用资源较少等，比较适合在嵌入式设备上面使用。SQLite 是 Android 系统自带的实现数据库功能的核心库，其代码实现分为 Java 和 C 两个部分，Java 部分的代码在/frameworks/base/core/java/android/database 中，目录结构如下所示。

```
frameworks/base/core/java/android/database/
├── sqlite                                //SQLite 的框架文件
├── AbstractCursor.java                   //游标的抽象类
├── AbstractWindowedCursor.java
├── BulkCursorDescriptor.java
├── BulkCursorNative.java
├── BulkCursorToCursorAdaptor.java        //游标适配器
├── CharArrayBuffer.java
├── ContentObservable.java
├── ContentObserver.java                  //内容观察者
├── CrossProcessCursor.java
├── CrossProcessCursorWrapper.java        //CrossProcessCursor 的封装类
├── Cursor.java                           //游标实现类
├── CursorIndexOutOfBoundsException.java  //游标出界异常
├── CursorJoiner.java
├── CursorToBulkCursorAdaptor.java        //适配器
├── CursorWindow.java                     //游标窗口
├── CursorWindowAllocationException.java  //游标窗口异常
├── CursorWrapper.java                    //游标封装类
├── DatabaseErrorHandler.java             //数据库错误句柄
├── DatabaseUtils.java                    //数据库工具类
```



```
├── DataSetObservable.java
├── DataSetObserver.java
├── DefaultDatabaseErrorHandler.java          //默认数据库错误句柄
├── IBulkCursor.java
├── IContentObserver.aidl                      //aidl 用于跨进程通信
├── MatrixCursor.java
├── MergeCursor.java
├── Observable.java
├── package.html
├── SQLException.java                          //数据库异常
└── StaleDataException.java
```

Java 层的代码主要是实现 SQLite 的框架和接口的实现，方便用户开发应用程序时能简单地操作数据库，并且捕获数据库异常。

C++层的代码在/external/sqlite 路径下，其目录结构如下所示。

```
external/sqlite/
├── android                                    //Android 数据库的一些工具包
└── dist                                       //Android 数据库底层实现
```

从上面 Java 和 C 部分的代码目录结构可以看出，SQLite 在 Android 中还是有很重要的地位的，并且在 SDK 中会有开放的接口让应用程序可以很简单方便地操作数据库，对数据进行存储和删除。

1.2.6　系统运行库部分

众所周知，Android 系统的应用层是采用 Java 开发的，由于 Java 语言的跨平台特性，Java 代码必须运行在虚拟机中。正是因为这个特性，Android 系统也自己实现了一个类似 JVM 但是更适用于嵌入式平台的 Java 虚拟机，这被称为 Dalvik。

Dalvik 功能等同于 JVM，为 Android 平台上的 Java 代码提供了运行环境，Dalvik 本身是由 C++语言实现的，在源码中根目录下有 dalvik 文件夹，里面存放的是 Dalvik 虚拟机的实现代码，其目录结构如下所示。

```
./
├── dalvikvm                                   //入口目录
├── dexdump                                    //DEX 反汇编
├── dexgen                                     //DEX 生成相关
├── dexlist                                    //DEX 列表
├── dexopt                                     //优化
├── docs                                       //文档
├── dvz                                        //Zygot 相关
├── dx                                         //DX 工具，将多个 Java 转换为 DEX
├── hit
├── libdex                                     //DEX 库的实现代码
├── opcode-gen
├── tests                                      //测试相关
├── tools                                      //工具
├── unit-tests                                 //测试相关
├── vm                                         //虚拟机的实现
```

```
├── Android.mk                          //Makefile
├── CleanSpec.mk
├── MODULE_LICENSE_APACHE2
├── NOTICE
└── README.txt
```

正是有上面这些代码实现的 Android 虚拟机，所以应用程序生成的二进制执行文件能够快速、稳定地运行在 Android 系统上。

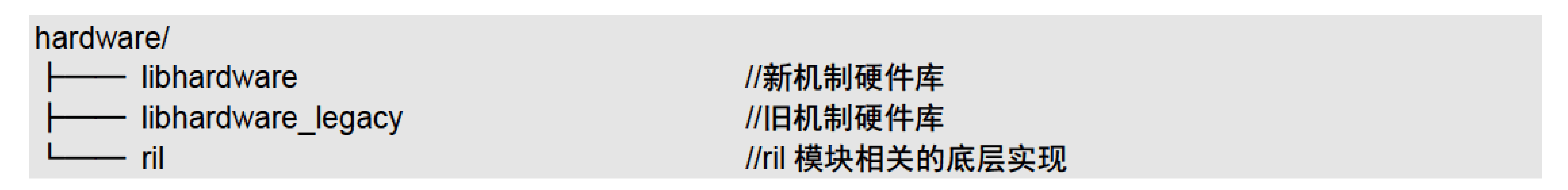
而从 Android 5.0 开始，Android 应用程序的默认运行环境为 ART，ART 模式拥有更快更高的运行效率。其目录结构如图 1-16 所示。



图 1-16　ART 模块的目录结构

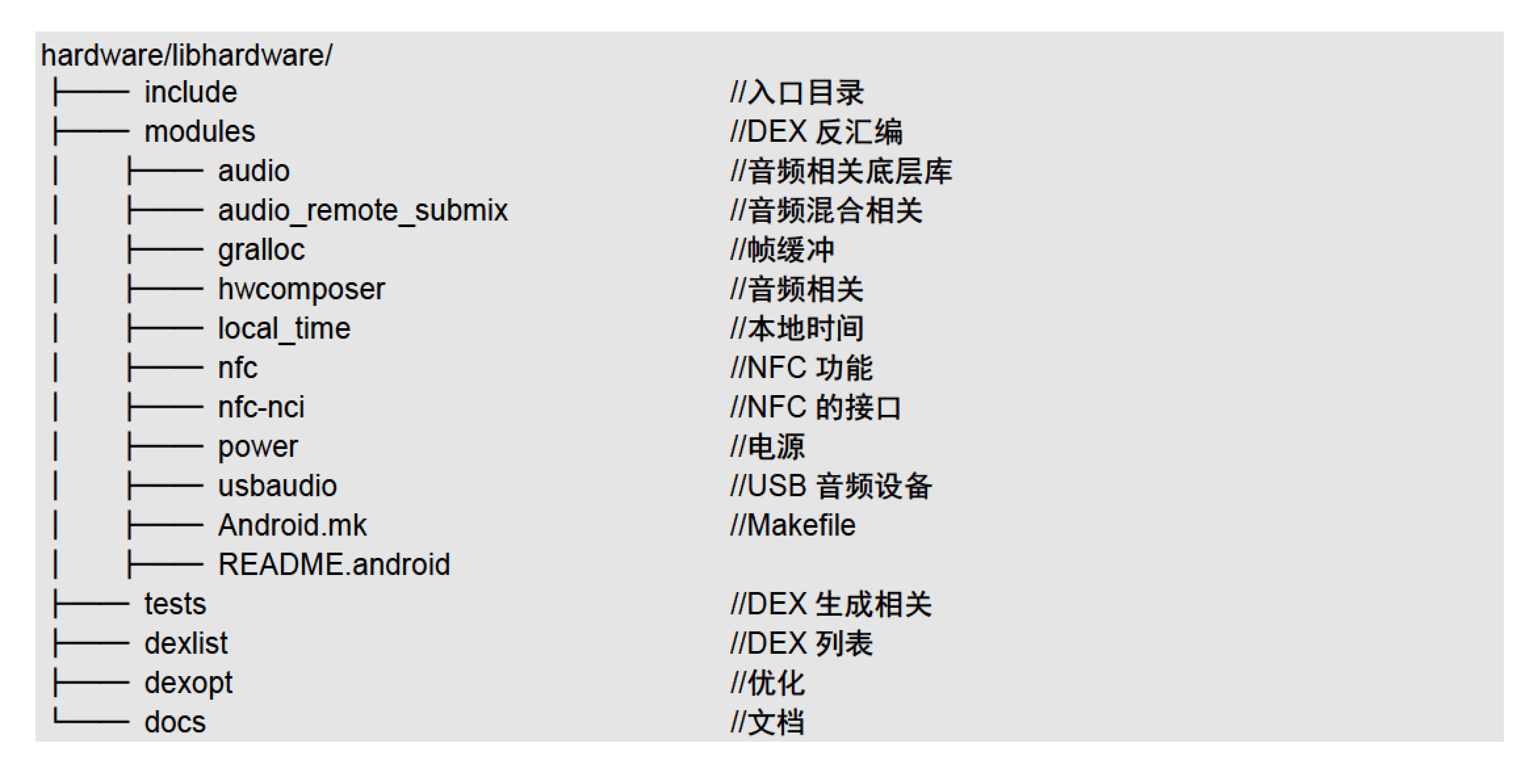
1.2.7　硬件抽象层部分

Android 的硬件抽象层是各种功能的底层实现，理论上不同的硬件平台会有不同的硬件抽象层实现，这一个层次也是与驱动层和硬件层有紧密联系的，起着承上启下的作用，对上要实现应用程序框架层的接口，对下要实现一些硬件基本功能以及调用驱动层的接口。需要注意的是，这一层也是广大 OEM 厂商改动最大的一层，因为这一层的代码和终端采用什么样硬件的硬件平台有很大关系。源码中存放的是硬件抽象层框架的实现代码和一些平台无关性的接口的实现。硬件抽象层代码在源码根目录下的 hardware 文件夹中，其目录结构如下所示。



```
hardware/
├── libhardware                         //新机制硬件库
├── libhardware_legacy                  //旧机制硬件库
└── ril                                 //ril 模块相关的底层实现
```

从上面的目录结构可以看出，硬件抽象层中主要实现了一些底层的硬件库，用来实现应用层框架层中的功能，具体硬件库中有哪些内容，可以继续细分其目录结构，例如，libhardware 目录下的结构为：



```
hardware/libhardware/
├── include                             //入口目录
├── modules                             //DEX 反汇编
|    ├── audio                          //音频相关底层库
|    ├── audio_remote_submix            //音频混合相关
|    ├── gralloc                        //帧缓冲
|    ├── hwcomposer                     //音频相关
|    ├── local_time                     //本地时间
|    ├── nfc                            //NFC 功能
|    ├── nfc-nci                        //NFC 的接口
|    ├── power                          //电源
|    ├── usbaudio                       //USB 音频设备
|    ├── Android.mk                     //Makefile
|    ├── README.android
├── tests                               //DEX 生成相关
├── dexlist                             //DEX 列表
├── dexopt                              //优化
└── docs                                //文档
```

从上面的目录结构可以分析出，libhardware 目录主要是 Android 系统的某些功能的底层实现，包括 audio、nfc 和 power。

而 libhardware_legacy 的目录与 libhardware 大同小异，只是针对旧的实现方式做的一套硬件库，其目录下还有 uevent、wifi 以及虚拟机的底层实现。这两个目录下的代码一般会由设备厂家根据自身的硬件平台来实现符合 Android 机制的硬件库。

而 ril 目录下存放的是无线硬件设备与电话的实现，其目录结构如下所示。



```
hardware/ril/
├── include                                   //头文件
├── libril                                    //libril 库
├── mock-ril
├── reference-ril                             //reference ril 库
├── rild                                      //ril 守护进程
└── CleanSpec.mk
```

1.3　分析源码中提供的接口

Android 源码当中提供了很多资源、工具或者文档供开发者使用，当然其中也包括应用程序开发接口的实现，也就是开发应用程序所使用的 SDK 的 API。正是由于有了这些种类丰富、功能强大、抽象程度高的接口，才让开发应用程序变得简单方便。本节将详细讲解 Android 系统中这些接口的基本知识。

1.3.1　暴露接口和隐藏接口

从本书前面的内容可知，可以从源码中自己编译生成一个 SDK，其功能作用等同于官网上的单独下载的 SDK 开发包。这说明源码中有 SDK 的实现代码，不仅可以提供与独立 SDK 相同的 API 接口，而且会有一些 SDK 开发包中不具备的 API 接口。当然，这部分隐藏的接口在基于 SDK 开发时是看不到的，只有在基于源码开发或者往独立的 SDK 中“增加”隐藏接口时才能调用到。

究竟源码中的哪些接口是暴露接口，哪些接口是隐藏接口呢？例如，要做一个 APP 来统计电量消耗信息，以及 Wi-Fi 或者蓝牙的打开时间，在 SDK 中是没有直接相关的接口来调用的。当然通过其他途径可以找到很多种方法来满足这个需求，但此处主要讲解怎么用源码中的隐藏接口来实现这些功能。

在源码路径/frameworks/base/core/java/android/os 目录下，存在两个电池相关的文件，即 BatteryStats.java 和 BatteryStatsImpl.java，前者声明了一个电池相关的抽象类 BatteryStats，后者继承了 BatteryStats，并实现了里面的方法。下面来看文件 BatteryStats.java 中的抽象类，在这个类中定义了很多变量来记录系统功能的状态变化，例如：

☑　Wi-Fi 开关。
☑　蓝牙开关。
☑　音频打开。
☑　视频打开。
☑　上次充电时刻。

上述信息用来计算各个模块的电量消耗情况，同时在里面也定义了其他的抽象类，里面的抽象接口都可以用来计算电量消耗，文件 BatteryStats.java 的代码如下所示。

```
public abstract class BatteryStats implements Parcelable {
    /*省略部分代码*/
    //Wi-Fi 开启时间
    public static final int WIFI_RUNNING = 4;
    //Wi-Fi 完全锁定时间
    public static final int FULL_WIFI_LOCK = 5;
    //Wi-Fi 扫描时间
    public static final int WIFI_SCAN = 6;
    //Wi-Fi 其他功能开启时间
    public static final int WIFI_MULTICAST_ENABLED = 7;
    //音频开启时间
    public static final int AUDIO_TURNED_ON = 7;
    //视频开启时间
    public static final int VIDEO_TURNED_ON = 8;
    //系统状态自从上次变化到现在
    public static final int STATS_SINCE_CHARGED = 0;
    //上一次的系统状态
    public static final int STATS_LAST = 1;
    //现在的系统状态
    public static final int STATS_CURRENT = 2;
    //从上次拔下设备到现在的状态
    public static final int STATS_SINCE_UNPLUGGED = 3;
    /*省略部分代码*/
    public static abstract class Uid {
        //得到相关联 UID 锁屏状态
        public abstract Map<String, ? extends Wakelock> getWakelockStats();
        public static abstract class Wakelock {
            public abstract Timer getWakeTime(int type);
        }
        //得到相关联 UID 的传感器状态
        public abstract Map<Integer, ? extends Sensor> getSensorStats();
        //得到 Pid 状态
        public abstract SparseArray<? extends Pid> getPidStats();
        //得到进程状态
        public abstract Map<String, ? extends Proc> getProcessStats();
        //得到包状态
        public abstract Map<String, ? extends Pkg> getPackageStats();
        /**
          *  得到 Uid
          * {@hide}
          */
        public abstract int getUid();
        /**
          *  得到 Tcp 接收到的字节数
          * {@hide}
          */
        public abstract long getTcpBytesReceived(int which);
```

```
/**
 * 得到 Tcp 发出的字节数
 * {@hide}
 */
public abstract long getTcpBytesSent(int which);
//记录 Wi-Fi 运行时刻
public abstract void noteWifiRunningLocked();
//记录 Wi-Fi 停止时刻
public abstract void noteWifiStoppedLocked();
public abstract void noteFullWifiLockAcquiredLocked();
public abstract void noteFullWifiLockReleasedLocked();
public abstract void noteWifiScanStartedLocked();
public abstract void noteWifiScanStoppedLocked();
public abstract void noteWifiMulticastEnabledLocked();
public abstract void noteWifiMulticastDisabledLocked();
public abstract void noteAudioTurnedOnLocked();
public abstract void noteAudioTurnedOffLocked();
public abstract void noteVideoTurnedOnLocked();
public abstract void noteVideoTurnedOffLocked();
//得到 Wi-Fi 运行时间
public abstract long getWifiRunningTime(long batteryRealtime, int which);
//得到 Wi-Fi 锁定时间
public abstract long getFullWifiLockTime(long batteryRealtime, int which);
//得到 Wi-Fi 扫描时间
public abstract long getWifiScanTime(long batteryRealtime, int which);
//得到 Wi-Fi 其他功能开启时间
public abstract long getWifiMulticastTime(long batteryRealtime, int which);
//得到音频开启时间
public abstract long getAudioTurnedOnTime(long batteryRealtime, int which);
//得到视频开启时间
public abstract long getVideoTurnedOnTime(long batteryRealtime, int which);
static final String[] USER_ACTIVITY_TYPES = {
    "other", "button", "touch"
};
public static final int NUM_USER_ACTIVITY_TYPES = 3;
public abstract void noteUserActivityLocked(int type);
public abstract boolean hasUserActivity();
public abstract int getUserActivityCount(int type, int which);
//传感器抽象类
public static abstract class Sensor {
public static final int GPS = -10000;
    public abstract int getHandle();
    public abstract Timer getSensorTime();
}
//Pid 类
public class Pid {
    public long mWakeSum;
    public long mWakeStart;
}
//进程相关类
public static abstract class Proc {
```

```
            public static class ExcessivePower {
                //唤醒方式
                public static final int TYPE_WAKE = 1;
                //CPU 类型
                public static final int TYPE_CPU = 2;
                public int type;
                public long overTime;
                public long usedTime;
            }
            //得到用户时间
            public abstract long getUserTime(int which);
            //得到系统时间
            public abstract long getSystemTime(int which);
            //得到状态
            public abstract int getStarts(int which);
            //得到 CPU 在前台运行的时间
            public abstract long getForegroundTime(int which);
            //得到 CPU 的速度等级
            public abstract long getTimeAtCpuSpeedStep(int speedStep, int which);
            //得到剩余的电量
            public abstract int countExcessivePowers();
            public abstract ExcessivePower getExcessivePower(int i);
        }
    }
/*省略部分代码*/
/**  得到屏幕点亮的时间
     * {@hide}
     */
    public abstract long getScreenOnTime(long batteryRealtime, int which);
/**  根据屏幕点亮的等级，得到相应时间
     * {@hide}
     */
    public abstract long getScreenBrightnessTime(int brightnessBin, long batteryRealtime, int which);
/**
     *  得到用电池的时候电话运行时的时间
     * {@hide}
     */
    public abstract long getPhoneOnTime(long batteryRealtime, int which);
/**
*  得到手机处于不同信号强度的时间
     * {@hide}
     */
    public abstract long getPhoneSignalStrengthTime(int strengthBin,
            long batteryRealtime, int which);
/**
     *  得到手机扫描信号用掉的时间
     * {@hide}
     */
    public abstract long getPhoneSignalScanningTime(long batteryRealtime, int which);
/**
     *  得到手机扫描到不同信号强度用掉的时间
```

```
     * {@hide}
     */
    public abstract int getPhoneSignalStrengthCount(int strengthBin, int which);
    /**
     * 得到手机不同数据连接消耗的时间
     * {@hide}
     */
    public abstract long getPhoneDataConnectionTime(int dataType,
            long batteryRealtime, int which);
/**
     * 得到手机进入到不同数据连接所消耗的时间
     * {@hide}
     */
    public abstract int getPhoneDataConnectionCount(int dataType, int which);
     /**
     * 得到手机处于 Wi-Fi 打开状态的时间
     * {@hide}
     */
    public abstract long getWifiOnTime(long batteryRealtime, int which);
    /**
     * 得到手机处于 Wi-Fi 打开状态并且驱动层的 Wi-Fi 也处于打开状态时的时间
     * {@hide}
     */
    public abstract long getGlobalWifiRunningTime(long batteryRealtime, int which);
    /**
     * 得到手机蓝牙处于打开状态时的时间
     * {@hide}
     */
    public abstract long getBluetoothOnTime(long batteryRealtime, int which);
}
```

从上面的源代码可以看出，在文件 BatteryStats.java 中定义了很多与电池电量和系统状态相关的函数和变量，有一些函数在声明时加上了@hide 字样，说明这些接口因为不稳定或者其他方面的原因暂时被 Google 隐藏了，不能通过 SDK 进行访问，可能会在以后的版本中开放。

对于没有标记@hide 接口的函数，则不属于 Google 官方声明的隐藏接口，但是同样可以将其看成是隐藏的，只不过 Google 短期内或者一直都不会将其开放，所以不会特别进行维护，其形式与有@hide 的隐藏接口一致。

介绍完文件 BatteryStats.java 中定义的隐藏接口后，接下来介绍文件 BatteryStatsImpl.java，在此文件中继承了 BatteryStats 抽象类，对 BatteryStats 中的很多隐藏方法进行了实现，其具体代码如下所示。

```
public final class BatteryStatsImpl extends BatteryStats {
/*省略了部分代码*/
        @Override
        public int getUid() {
            return mUid;
        }
        @Override
        public long getTcpBytesReceived(int which) {
            if (which == STATS_LAST) {
```

```
                return mLoadedTcpBytesReceived;
            } else {
                long current = computeCurrentTcpBytesReceived();
                if (which == STATS_SINCE_UNPLUGGED) {
                    current -= mTcpBytesReceivedAtLastUnplug;
                } else if (which == STATS_SINCE_CHARGED) {
                    current += mLoadedTcpBytesReceived;
                }
                return current;
            }
        }
        @Override
        public long getTcpBytesSent(int which) {
            if (which == STATS_LAST) {
                return mLoadedTcpBytesSent;
            } else {
                long current = computeCurrentTcpBytesSent();
                if (which == STATS_SINCE_UNPLUGGED) {
                    current -= mTcpBytesSentAtLastUnplug;
                } else if (which == STATS_SINCE_CHARGED) {
                    current += mLoadedTcpBytesSent;
                }
                return current;
            }
        }
    @Override public long getScreenOnTime(long batteryRealtime, int which) {
        return mScreenOnTimer.getTotalTimeLocked(batteryRealtime, which);
    }
    @Override public long getScreenBrightnessTime(int brightnessBin,
            long batteryRealtime, int which) {
        return mScreenBrightnessTimer[brightnessBin].getTotalTimeLocked(
                batteryRealtime, which);
    }
    @Override public long getPhoneOnTime(long batteryRealtime, int which) {
        return mPhoneOnTimer.getTotalTimeLocked(batteryRealtime, which);
    }
@Override public long getPhoneSignalStrengthTime(int strengthBin,
            long batteryRealtime, int which) {
        return mPhoneSignalStrengthsTimer[strengthBin].getTotalTimeLocked(
                batteryRealtime, which);
    }
    @Override public long getPhoneSignalScanningTime(
            long batteryRealtime, int which) {
        return mPhoneSignalScanningTimer.getTotalTimeLocked(
                batteryRealtime, which);
    }
    @Override public int getPhoneSignalStrengthCount(int strengthBin, int which){
        return mPhoneSignalStrengthsTimer[strengthBin].getCountLocked(which);
    }
    @Override public long getPhoneDataConnectionTime(int dataType,
            long batteryRealtime, int which) {
```

```
        return mPhoneDataConnectionsTimer[dataType].getTotalTimeLocked(
                batteryRealtime, which);
    }
    @Override public int getPhoneDataConnectionCount(int dataType, int which) {
        return mPhoneDataConnectionsTimer[dataType].getCountLocked(which);
    }
    @Override public long getWifiOnTime(long batteryRealtime, int which) {
        return mWifiOnTimer.getTotalTimeLocked(batteryRealtime, which);
    }
    @Override public long getGlobalWifiRunningTime(long batteryRealtime, int which) {
        return mGlobalWifiRunningTimer.getTotalTimeLocked(batteryRealtime, which);
    }
    @Override public long getBluetoothOnTime(long batteryRealtime, int which) {
        return mBluetoothOnTimer.getTotalTimeLocked(batteryRealtime, which);
    }
}
```

在上述代码中，BatteryStatsImpl 继承了类 BatteryStats，然后实现了其中的隐藏接口，这样当进行应用程序开发时就可以使用这些还未开放的隐藏接口了，具体使用隐藏接口的方法将在 1.3.2 节中详细介绍。

1.3.2　调用隐藏接口

在 1.3.1 节的内容中，以 BatteryStats 这个类为例详细分析了 Android 中存在的隐藏接口。下面将分析在源码中使用这些隐藏接口的方法，然后分析在应用程序开发过程中调用隐藏接口的方法。

Android 系统中在 Settings 程序中使用类 BatteryStats，主要用来统计一些系统功能模块的工作时间和耗电情况。Settings 中使用 BatteryStats 类的是文件 PowerUsageSummary.java，其存放路径为 /packages/apps/settings/src/com/android/settings/fuelgauge。

文件 PowerUsageSummary.java 的主要功能是统计系统中的电量信息，在此用到了一些 BatteryStats 类中的隐藏接口，下面具体分析其实现代码，其中部分典型代码如下所示。

```
public class PowerUsageSummary extends PreferenceFragment implements Runnable {
    /*省略部分代码*/
    //定义了 BatteryStats 相关的 3 个对象，涉及跨进程通信
    private static BatteryStatsImpl sStatsXfer;
    IBatteryStats mBatteryInfo;
    BatteryStatsImpl mStats;
    //在 onCreate 中初始化 mBatteryInfo 对象
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        if (icicle != null) {
            //对 mStats 对象赋值
            mStats = sStatsXfer;
        }
        addPreferencesFromResource(R.xml.power_usage_summary);
        //初始化 mBatteryInfo 对象
        mBatteryInfo = IBatteryStats.Stub.asInterface(
```

```
                    ServiceManager.getService("batteryinfo"));
        mUm =(UserManager)getActivity().getSystemService(Context.USER_SERVICE);
        mAppListGroup = (PreferenceGroup) findPreference(KEY_APP_LIST);
        mBatteryStatusPref = mAppListGroup.findPreference(KEY_BATTERY_STATUS);
        mPowerProfile = new PowerProfile(getActivity());
        setHasOptionsMenu(true);
    }

        private void addPhoneUsage(long uSecNow) {
        long phoneOnTimeMs = mStats.getPhoneOnTime(uSecNow, mStatsType) / 1000;
        double phoneOnPower = mPowerProfile.getAveragePower(PowerProfile.POWER_RADIO_ACTIVE)
                                                            * phoneOnTimeMs / 1000;
        addEntry(getActivity().getString(R.string.power_phone), DrainType.PHONE,
  phoneOnTimeMs,
                              R.drawable.ic_settings_voice_calls, phoneOnPower);
    }
}
    //当 mStats 没有初始化时则通过 Parcel 接口继续初始化
    private void load() {
        try {
            byte[] data = mBatteryInfo.getStatistics();
            Parcel parcel = Parcel.obtain();
            parcel.unmarshall(data, 0, data.length);
            parcel.setDataPosition(0);
            mStats = com.android.internal.os.BatteryStatsImpl.CREATOR.createFromParcel(parcel);
            mStats.distributeWorkLocked(BatteryStats.STATS_SINCE_CHARGED);
        } catch (RemoteException e) {
            Log.e(TAG, "RemoteException:", e);
        }
    }

    //得到屏幕的使用时间和耗电情况
    private void addScreenUsage(long uSecNow) {
        double power = 0;
        // getScreenOnTime()为 BatteryStats 中的隐藏接口
        long screenOnTimeMs = mStats.getScreenOnTime(uSecNow, mStatsType) / 1000;
        power += screenOnTimeMs *
                mPowerProfile.getAveragePower(PowerProfile.POWER_SCREEN_ON);
        final double screenFullPower =
                mPowerProfile.getAveragePower(PowerProfile.POWER_SCREEN_FULL);
        for (int i = 0; i < BatteryStats.NUM_SCREEN_BRIGHTNESS_BINS; i++) {
            double screenBinPower = screenFullPower * (i + 0.5f)
                    / BatteryStats.NUM_SCREEN_BRIGHTNESS_BINS;
            // getScreenBrightnessTime 为 BatteryStats 中的隐藏接口
            long brightnessTime = mStats.getScreenBrightnessTime(i, uSecNow, mStatsType) / 1000;
            power += screenBinPower * brightnessTime;
            if (DEBUG) {
                Log.i(TAG, "Screen bin power = " + (int) screenBinPower +
                                        ", time = " + brightnessTime);
            }
```

```
        }
        power /= 1000;
        addEntry(getActivity().getString(R.string.power_screen),
                                    DrainType.SCREEN, screenOnTimeMs,
                                    R.drawable.ic_settings_display, power);
    }
    //得到 Wi-Fi 的使用时间和耗电情况
    private void addWiFiUsage(long uSecNow) {
        //getWifiOnTime 为 BatteryStats 中的隐藏接口
        long onTimeMs = mStats.getWifiOnTime(uSecNow, mStatsType) / 1000;
        //getGlobalWifiRunningTime 为 BatteryStats 中的隐藏接口
        long runningTimeMs = mStats.getGlobalWifiRunningTime(uSecNow, mStatsType) / 1000;
        if (DEBUG) Log.i(TAG, "WIFI runningTime=" + runningTimeMs
                + " app runningTime=" + mAppWifiRunning);
        runningTimeMs -= mAppWifiRunning;
        if (runningTimeMs < 0) runningTimeMs = 0;
        double wifiPower = (onTimeMs * 0 /* TODO */
                    * mPowerProfile.getAveragePower(PowerProfile.POWER_WIFI_ON)
                                                            + runningTimeMs *
            mPowerProfile.getAveragePower(PowerProfile.POWER_WIFI_ON)) / 1000;
        if (DEBUG) Log.i(TAG, "WIFI power=" + wifiPower + " from procs=" + mWifiPower);
        BatterySipper bs=addEntry(getActivity().getString(R.string.power_wifi), DrainType.WIFI,
                        runningTimeMs, R.drawable.ic_settings_wifi, wifiPower + mWifiPower);
        aggregateSippers(bs, mWifiSippers, "WIFI");
    }
    //得到蓝牙的使用时间和耗电情况
    private void addBluetoothUsage(long uSecNow) {
        //getBluetoothOnTime 为 BatteryStats 中的隐藏接口
        long btOnTimeMs = mStats.getBluetoothOnTime(uSecNow, mStatsType) / 1000;
                                                double btPower = btOnTimeMs *
                mPowerProfile.getAveragePower(PowerProfile.POWER_BLUETOOTH_ON)
                                                                        / 1000;
       //getBluetoothPingCount 为 BatteryStats 中的隐藏接口
        int btPingCount = mStats.getBluetoothPingCount();
        btPower += (btPingCount *
mPowerProfile.getAveragePower(PowerProfile.POWER_BLUETOOTH_AT_CMD)) / 1000;
        BatterySipper bs =
                addEntry(getActivity().getString(R.string.power_bluetooth),
                                        DrainType.BLUETOOTH, btOnTimeMs,
                                        R.drawable.ic_settings_bluetooth,
                                        btPower + mBluetoothPower);
        aggregateSippers(bs, mBluetoothSippers, "Bluetooth");
    }
```

从上述部分代码可以看出，在一些函数中广泛用到了 BatteryStats 类中的隐藏接口。首先调用隐藏接口来获取模块使用时间，然后得到平均用电时间，经过一定的算法即可得出该模块的耗电情况。如果没有这样的隐藏接口，计算耗电时间会是很麻烦的一件事，而且可能会用到源码中更多没有权限去调用的代码，开发难度将会大大增加。

1.4 编译源码

编译 Android 源码的方法非常简单，只需使用 Android 源码根目录下的 Makefile，执行 make 命令即可实现。当然在编译 Android 源码之前，首先要确定已经完成同步工作。进入 Android 源码目录使用 make 命令进行编译，使用此命令的格式如下所示。

```
$: cd ~/Android5.0（这里的 Android5.0 就是下载源码的保存目录）
$: make
```

编译 Android 源码可以得到~/project/android/cupcake/out 目录，笔者的截图界面如图 1-17 所示。

```
主意：external/doclava/src/com/google/doclava/Stubs.java 使用了未经检查或不安全
约操作。
主意：要了解详细信息，请使用 -Xlint:unchecked 重新编译。
otice file: external/doclava/NOTICE -- out/host/linux-x86/obj/NOTICE_FILES/src/
/framework/doclava.jar.txt
nstall: out/host/linux-x86/framework/doclava.jar
arget Java: core (out/target/common/obj/JAVA_LIBRARIES/core_intermediates/class
es)
主意：某些输入文件使用或覆盖了已过时的 API。
主意：要了解详细信息，请使用 -Xlint:deprecation 重新编译。
主意：某些输入文件使用了未经检查或不安全的操作。
主意：要了解详细信息，请使用 -Xlint:unchecked 重新编译。
opying: out/target/common/obj/JAVA_LIBRARIES/core_intermediates/classes-jarjar.
jar
opying: out/target/common/obj/JAVA_LIBRARIES/core_intermediates/emma_out/lib/cl
asses-jarjar.jar
opying: out/target/common/obj/JAVA_LIBRARIES/core_intermediates/classes.jar
arget Java: conscrypt (out/target/common/obj/JAVA_LIBRARIES/conscrypt_intermedi
ates/classes)
主意：某些输入文件使用或覆盖了已过时的 API。
主意：要了解详细信息，请使用 -Xlint:deprecation 重新编译。
主意：某些输入文件使用了未经检查或不安全的操作。
主意：要了解详细信息，请使用 -Xlint:unchecked 重新编译。
ost Prebuilt: jarjar (out/host/common/obj/JAVA_LIBRARIES/jarjar_intermediates/ avalib.jar)
nstall: out/host/linux-x86/framework/jarjar.jar
arJar: out/target/common/obj/JAVA_LIBRARIES/conscrypt_intermediates/classes-jarjar.jar
opying: out/target/common/obj/JAVA_LIBRARIES/conscrypt_intermediates/emma_out/lib/classes-jarjar.jar
opying: out/target/common/obj/JAVA_LIBRARIES/conscrypt_intermediates/classes.jar
arget Java: bouncycastle (out/target/common/obj/JAVA_LIBRARIES/bouncycastle_intermediates/classes)
主意：某些输入文件使用或覆盖了已过时的 API。
主意：要了解详细信息，请使用 -Xlint:deprecation 重新编译。
主意：某些输入文件使用了未经检查或不安全的操作。
主意：要了解详细信息，请使用 -Xlint:unchecked 重新编译。
arJar: out/target/common/obj/JAVA_LIBRARIES/bouncycastle_intermediates/classes-jarjar.jar
opying: out/target/common/obj/JAVA_LIBRARIES/bouncycastle_intermediates/emma_out/lib/classes-jarjar.jar
opying: out/target/common/obj/JAVA_LIBRARIES/bouncycastle_intermediates/classes.jar
arget Java: ext (out/target/common/obj/JAVA_LIBRARIES/ext_intermediates/classes)
```

图 1-17 编译过程的截图界面

整个编译过程也非常漫长，需要读者耐心等待。本节将详细讲解编译并在模拟器中运行 Android 5.0 源码的方法。

1.4.1 搭建编译环境

在编译 Android 源码之前，需要先进行环境搭建工作。下面以 Ubuntu 系统为例讲解搭建编译环境以及编译 Android 源码的方法。具体流程如下所示。

（1）安装 JDK，编译 Android 5.0 的源码需要 JDK 1.7，下载 jdk-7u22-linux-i586.bin 后进行安装，对应命令如下：

```
$ cd /usr
$ mkdir java
$ cd java
$ sudo cp jdk-7u22-linux-i586.bin
```

```
$ sudo chmod 755 jdk-7u22-linux-i586.bin
$ sudo sh jdk-7u22-linux-i586.bin
```

（2）设置 JDK 环境变量，将如下环境变量添加到主文件夹目录下的.bashrc 文件中，然后用 source 命令使其生效，加入的环境变量代码如下：

```
export JAVA_HOME=/usr/java/jdk1.7.0_27
export JRE_HOME=$JAVA_HOME/jre
export CLASSPATH=.:$JAVA_HOME/lib:$JRE_HOME/lib:$CLASSPATH
export PATH=$PATH:$JAVA_HOME/bin:$JAVA_HOME/bin/tools.jar:$JRE_HOME/bin
export ANDROID_JAVA_HOME=$JAVA_HOME
```

对于安装好的 JDK，并且在添加环境变量之后，可以输入并执行命令 java –version 来查看 JDK 的版本。如果有类似图 1-18 所示的信息输入，那么说明成功安装了 JDK。

```
OpenJDK Runtime Environment (IcedTea6 1.12.6) (6b27-1.12.6-1ubuntu0.10.04.2)
OpenJDK 64-Bit Server VM (build 20.0-b12, mixed mode)
xyh666168@wave-desktop:~$
```

图 1-18　执行命令 java –version 后

（3）安装需要的编译工具，对于 Linux 10.04 系统来说，只需要安装如下软件工具即可，在安装前保持计算机正常连接网络。

```
sudo apt-get install git-core gnupg flex bison gperf build-essential \
zip curl zlib1g-dev libc6-dev lib32ncurses5-dev ia32-libs \
x11proto-core-dev libx11-dev lib32readline5-dev lib32z-dev \
libgl1-mesa-dev g++-multilib mingw32 tofrodos python-markdown \
```

使用下面的命令做一个软链接文件。

```
sudo ln -s /usr/lib32/mesa/libGL.so.1 /usr/lib32/mesa/libGL.so
```

安装 11.10 系统需要的特别工具。

```
sudo apt-get install libx11-dev:i386
```

（4）开始设置高速缓存，目的是加快编译速度。对于配置不是很高的计算机来说，最好进行该设置，这样可以节约很多时间。设置方法是先用 vi 或者 gedit 软件打开宿主目录下的.bashrc 文件，然后在文件的最后添加如下的值。

```
export USE_CCACHE=1
```

保存后退出，重新登录系统以使设置生效，如图 1-19 所示。

在终端中切换到源码根目录中，然后执行下面的命令设置 ccache 的大小为 50G。

```
prebuilts/misc/linux-x86/ccache/ccache -M 50G
```

其实 ccache 就是一个执行文件，后面的-M 和 50G 是传递给 ccache 的参数，表示设置 50G 的缓存空间，此参数可以根据时间需要来修改。

（5）运行如下命令，导入编译 Android 源码所需的环境变量和其他参数。

```
source build/envsetup.sh
```

要想了解具体添加了哪些环境变量等，可以打开如图 1-20 所示的方框中对应的文件。

```
File  Edit  View  Terminal  Help
fi

# some more ls aliases
alias ll='ls -alF'
alias la='ls -A'
alias l='ls -CF'

# Alias definitions.
# You may want to put all your additions into a separate file like
# ~/.bash_aliases, instead of adding them here directly.
# See /usr/share/doc/bash-doc/examples in the bash-doc package.

if [ -f ~/.bash_aliases ]; then
    . ~/.bash_aliases
fi

# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if [ -f /etc/bash_completion ] && ! shopt -oq posix; then
    . /etc/bash_completion
fi
export USE_CCACHE=1
                                                    100,19        Bot
```

图 1-19　设置高速缓存

```
including device/samsung/manta/vendorsetup.sh
including device/generic/mips/vendorsetup.sh
including device/generic/armv7-a-neon/vendorsetup.sh
including device/generic/x86/vendorsetup.sh
including device/lge/hammerhead/vendorsetup.sh
including device/lge/mako/vendorsetup.sh
including device/asus/grouper/vendorsetup.sh
including device/asus/deb/vendorsetup.sh
including device/asus/tilapia/vendorsetup.sh
including device/asus/flo/vendorsetup.sh
including sdk/bash_completion/adb.bash
```

图 1-20　打开方框中对应的文件

（6）运行 lunch 命令选择编译目标，会出现一些已经预置好的项目。在此输入对应的数字，然后按 Enter 键确认选择编译目标对象，如图 1-21 所示。



图 1-21　选择编译目标

运行 lunch 命令并选择好编译目标后，接下来开始步入真正的编译阶段，运行如下命令进行编译。

```
make -j16
```

因为此处所用计算机的 CPU 是 i7 4770，所以使用 16。整个编译过程比较慢，因为计算机配置的问题，可能需要长时间等待。编译成功后会弹出如图 1-22 所示的提示信息。

```
File  Edit  View  Terminal  Help
+ MAKE_EXT4FS_CMD='make_ext4fs  -S out/target/product/generic/root/file_contexts -l 576716800 -a
system out/target/product/generic/obj/PACKAGING/systemimage_intermediates/system.img out/target/p
roduct/generic/system'
+ echo make_ext4fs -S out/target/product/generic/root/file_contexts -l 576716800 -a system out/ta
rget/product/generic/obj/PACKAGING/systemimage_intermediates/system.img out/target/product/generi
c/system
make_ext4fs -S out/target/product/generic/root/file_contexts -l 576716800 -a system out/target/pr
oduct/generic/obj/PACKAGING/systemimage_intermediates/system.img out/target/product/generic/syste
m
+ make_ext4fs -S out/target/product/generic/root/file_contexts -l 576716800 -a system out/target/
product/generic/obj/PACKAGING/systemimage_intermediates/system.img out/target/product/generic/sys
tem
Creating filesystem with parameters:
    Size: 576716800
    Block size: 4096
    Blocks per group: 32768
    Inodes per group: 7040
    Inode size: 256
    Journal blocks: 2200
    Label:
    Blocks: 140800
    Block groups: 5
    Reserved block group size: 39
Created filesystem with 1262/35200 inodes and 81850/140800 blocks
+ '[' 0 -ne 0 ']'
Install system fs image: out/target/product/generic/system.img
out/target/product/generic/system.img+ maxsize=588791808 blocksize=2112 total=576716800 reserve=5
947392
```

图 1-22　编译成功时的提示信息

在编译完成后，可以在源码中的 out/target/product/generic/目录下生成对应固件等文件，如图 1-23 所示。

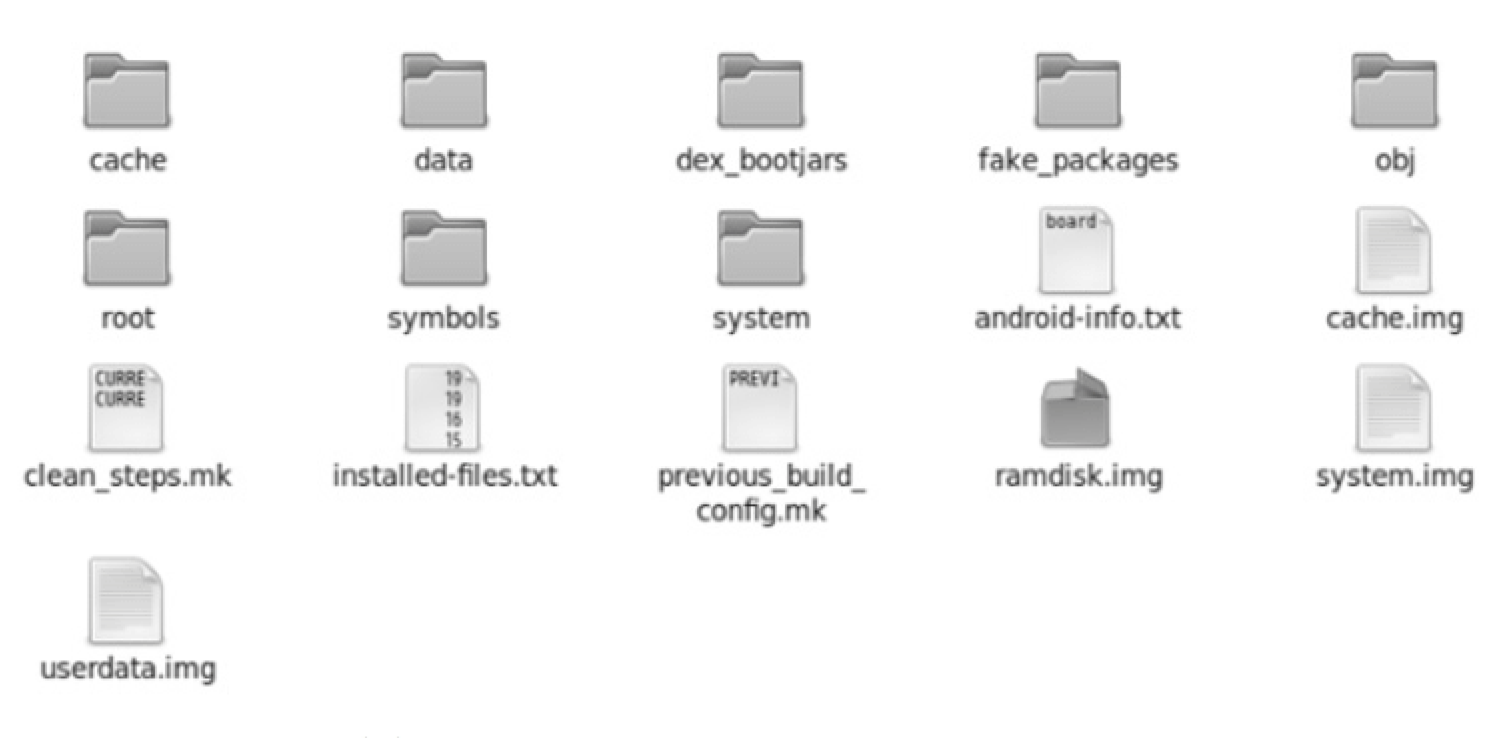


图 1-23　out/target/product/generic/目录

注意： 获取 Android 源码的过程是一个漫长的过程，一个疏忽就可能造成下载失败的结果。另外，编译 Android 源码的过程也是需要耐心等待的过程，为避免走弯路，可在网络中参考一些资源，例如，笔者就参考了网名为 xyh666168 的帖子，地址是 http://jingyan.baidu.com/article/a501d80ce61ad0ec630f5e0b.html。因为读者的机器配置是各种各样的，CPU 的型号参数也是不同的，所以建议读者多参考网络中的教程和解决方案。

1.4.2　在模拟器中运行

在模拟器中运行的步骤比较简单，只需在终端中执行下面的命令即可：

```
emulator
```

运行成功后的效果如图 1-24 所示。



图 1-24　在模拟器中的编译执行效果

1.5　编译源码生成 SDK

平时大部分 Android 应用程序开发是基于 SDK 实现的，其过程是使用 SDK 中的接口实现各种各样的功能。用户可以在 Android 的官方网站直接下载最新的 SDK 版本，也可以从源码中生成 SDK，因为源码中也包含 SDK 的代码。

下载的 Android 5.0 的源码的根目录下有一个 SDK 目录，所有与 SDK 相关的代码都存放在这个目录中，包括镜像文件、模拟器、ADB 等常用工具以及 SDK 中的开发包的文档，可以通过编译的方式生成开发需要的 SDK，编译命令如下所示。

```
$ Make SDK
```

当编译完成后，会在/out/host/linux-x86/sdk/目录下生成 SDK，这个 SDK 是完全与源码同步的，与官网上下载的 SDK 功能完全相同，会有开发用的 JAR 包，模拟器管理工具，ADB 调试工具，可以使用这个编译生成的 SDK 来开发应用程序。

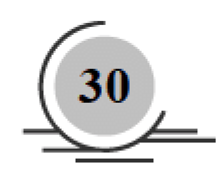
对于 Android 系统的开发，基本可以分为如下两种开发方式。

☑　基于 SDK 的开发。

☑　基于源码的开发。

在一般情况下，开发的应用程序都是基于 SDK 的开发，比较方便而且兼容性比较好。基于源码的开发相对于基于 SDK 的开发要求对源码的架构认识更深刻，一般用于需要修改系统层面的场合。两种方式应用场景不同，各有优缺点，本节将主要介绍基于 SDK 的开发。

如果想基于 SDK 开发 Android 的应用程序，需要 JDK、SDK 和一个开发环境，JDK 和 SDK 在不同的平台下有不同的版本，本章主要讨论 Windows 7 平台下的开发环境搭建。



（1）安装 JDK

由于 Android 的应用程序是使用 Java 语言开发的，所以首先需要安装 Java 的 JDK，下载地址为 http://java.sun.com/javase/downloads/index.jsp，选择合适的平台并下载最新版本的 JDK。

（2）安装 Eclipse

Eclipse 是开发 Android 应用程序的 IDE 环境，有非常丰富的插件可以使用，单击 http://www.eclipse.org/downloads/可以下载合适平台的最新版本 Eclipse。

（3）安装 Android SDK

Android SDK 是 Google 对外发布的专门用于 Android 开发的工具包，包括各种版本的开发框架和工具以及丰富的文档，打开 http://developer.android.com/sdk/index.html 可以下载最新版本的针对 Windows 7 平台的 SDK。

当下载完成上述 3 个工具之后，需要对开发环境进行如下配置。

（1）配置 Eclipse

第 1 步：打开 Eclipse，在菜单栏中选择 help | Install New Software 命令，出现如图 1-25 所示的窗口。

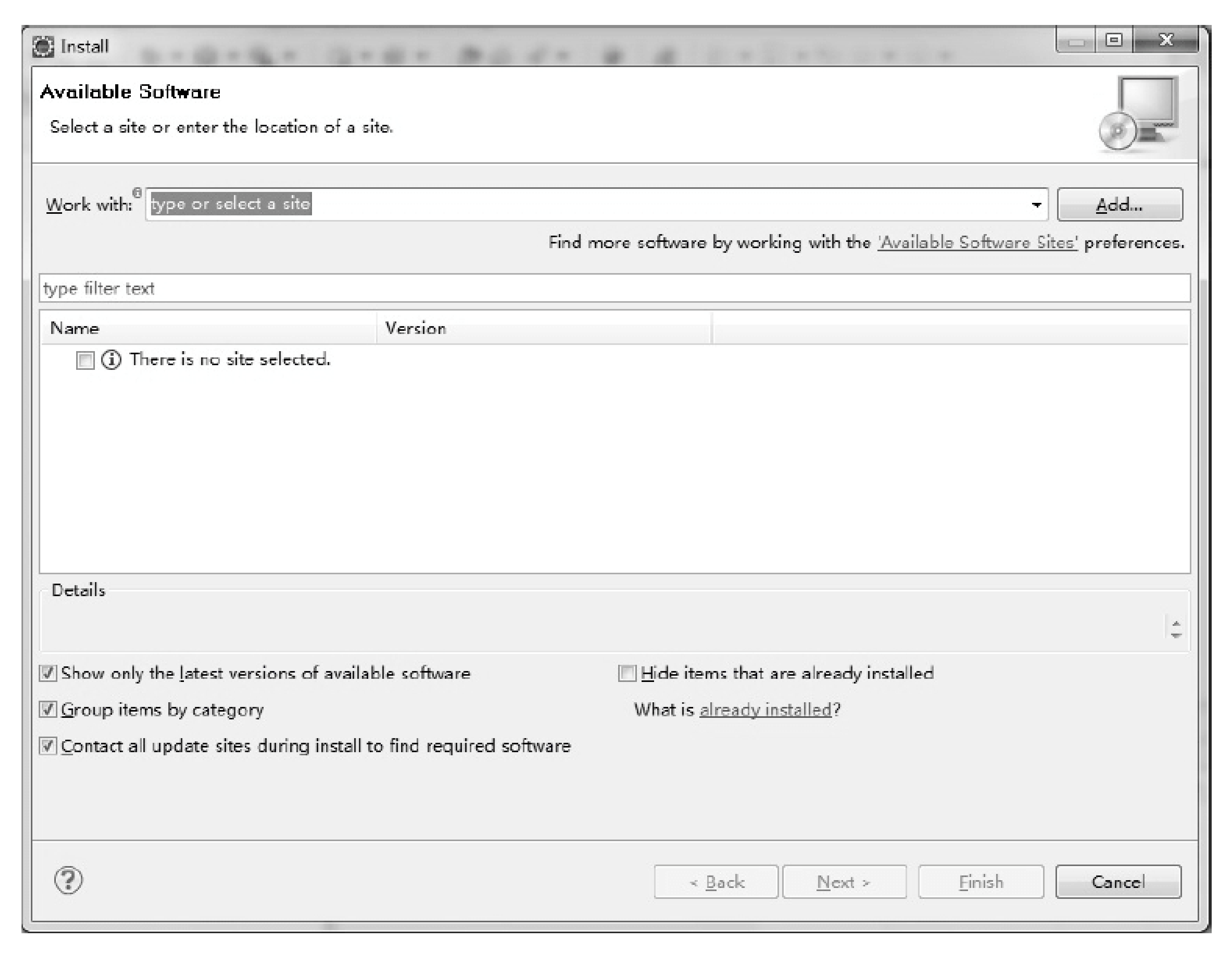


图 1-25　Install 窗口

第 2 步：单击 Add 按钮，会出现如图 1-26 所示的对话框。

第 3 步：在 Add Repository 对话框中可以新增一个站点，在 Name 文本框中输入 Android 或者自定义任何名字，在 Location 文本框中输入 https://dl-ssl.google.com/android/eclipse/，如图 1-27 所示。

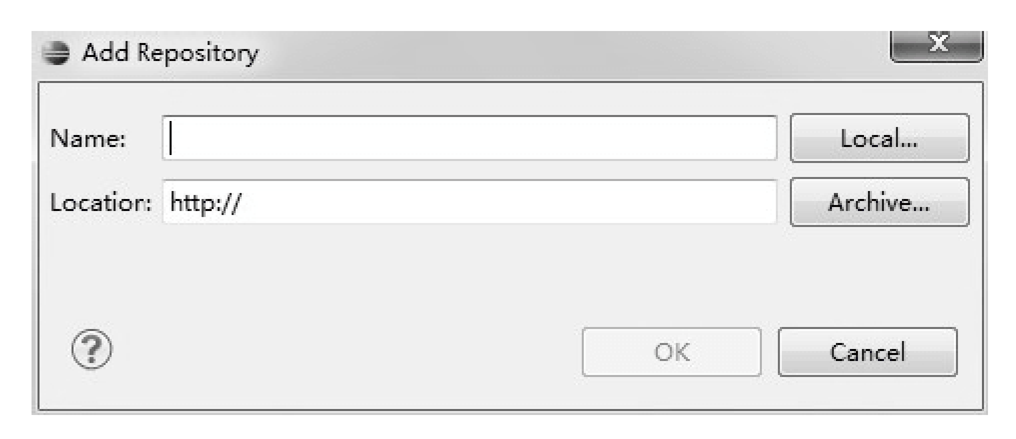


图 1-26　Add Repository 对话框

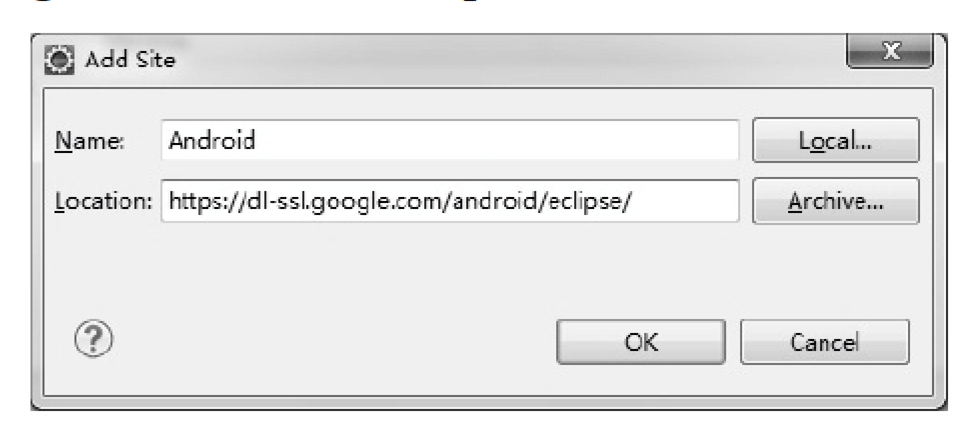


图 1-27　Add Site 对话框

第 4 步：如果发现 https://无法使用，可以改成 http://尝试一下，当输入好名字和地址之后，单击 OK 按钮，会出现如图 1-28 所示的窗口。

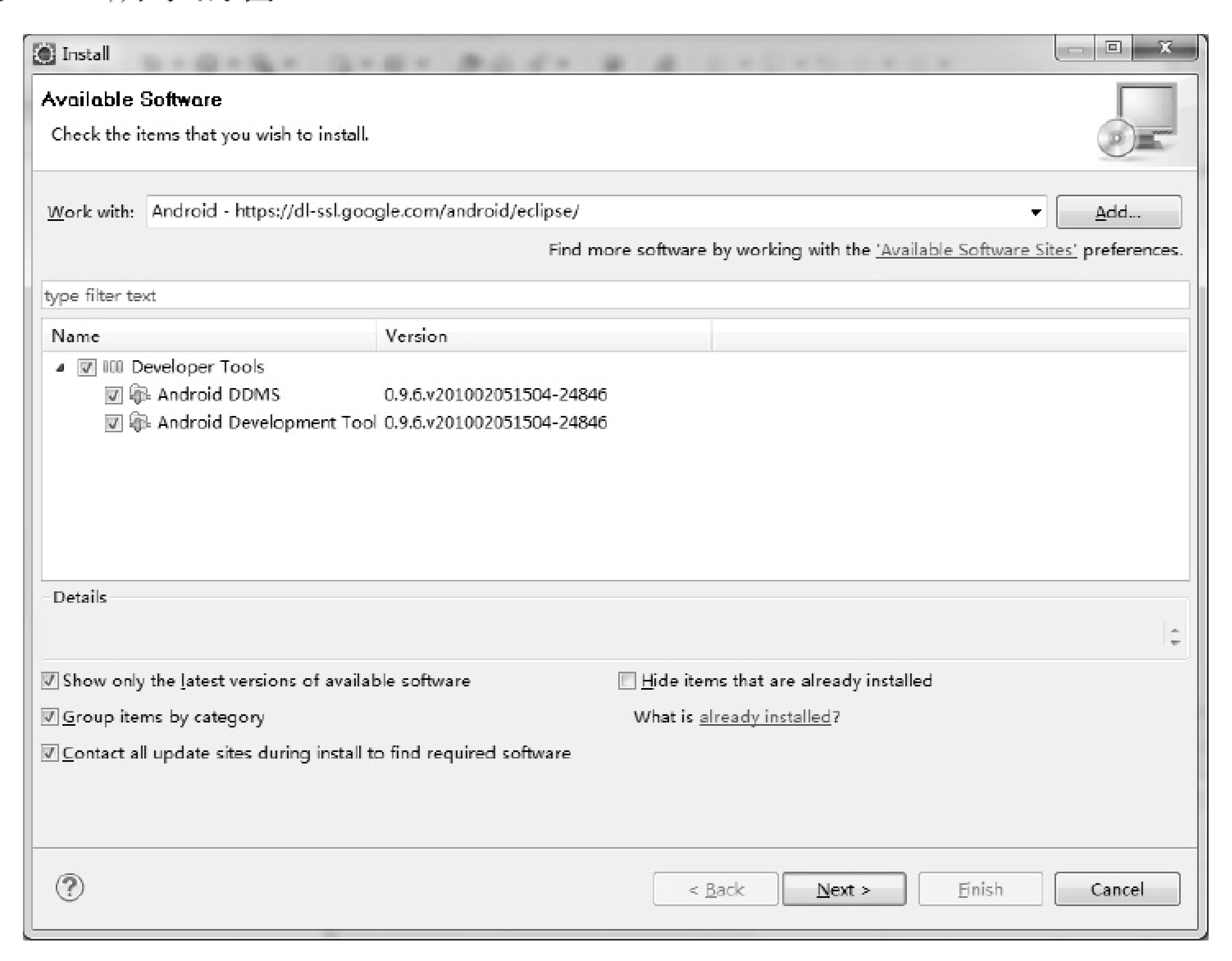


图 1-28　Install 窗口

图 1-28 中的两个插件都是开发 Android 必不可少的工具包，Android DDMS 是可以用来调试、管理 Android 进程、存储器、查看日志的工具，Android Development Tool 简称 ADT，是开发 Android 的插件，只有安装了 ADT 才能创建 Android 工程。

第 5 步：单击 Next 按钮，出现如图 1-29 所示的界面。

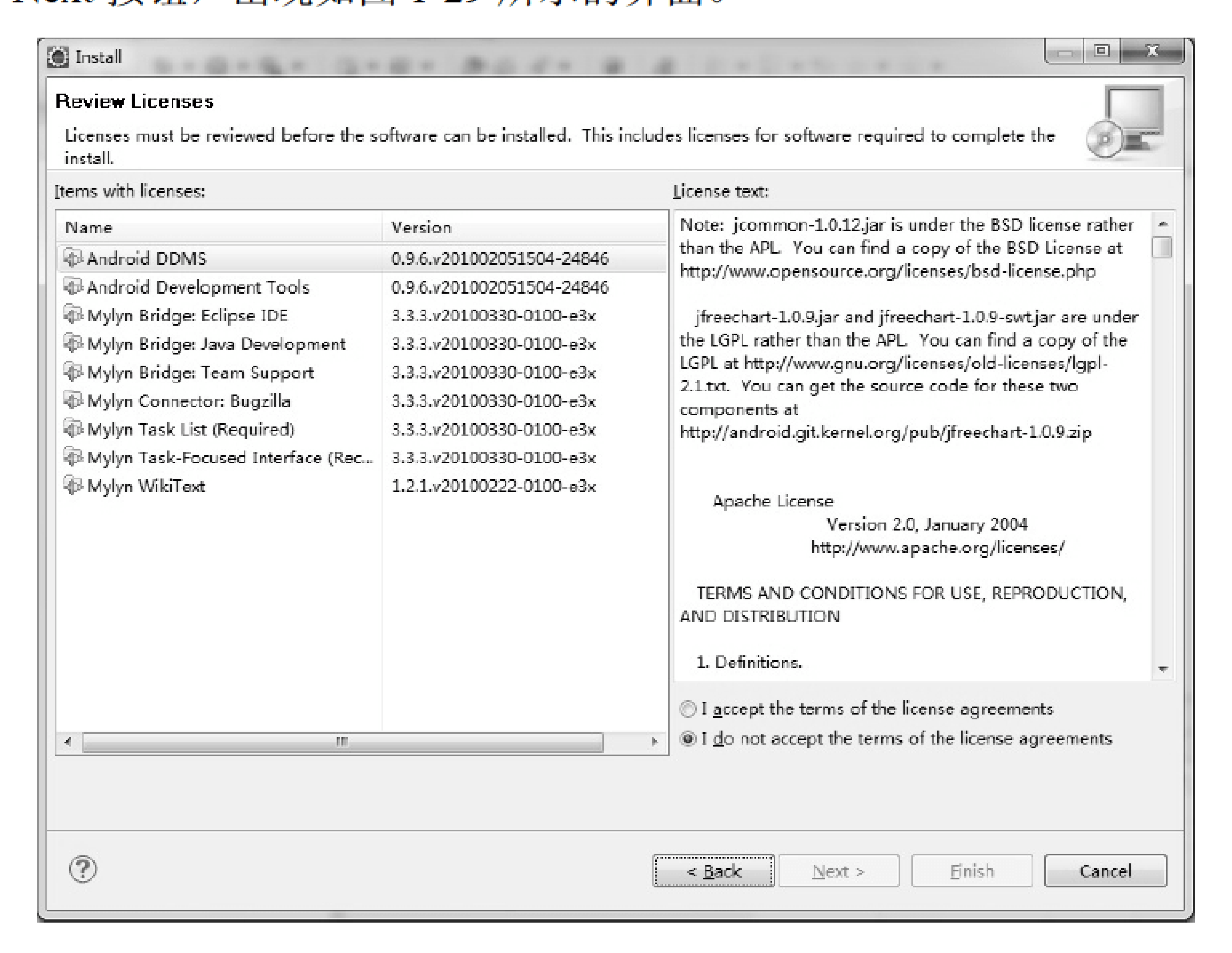
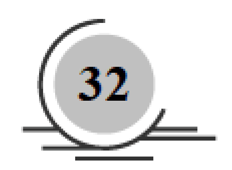


图 1-29　选择安装

在图 1-29 中列出了将会安装的工具包，选中 I accept…复选框，单击 Next 按钮会开始安装插件，界



面如图 1-30 所示。

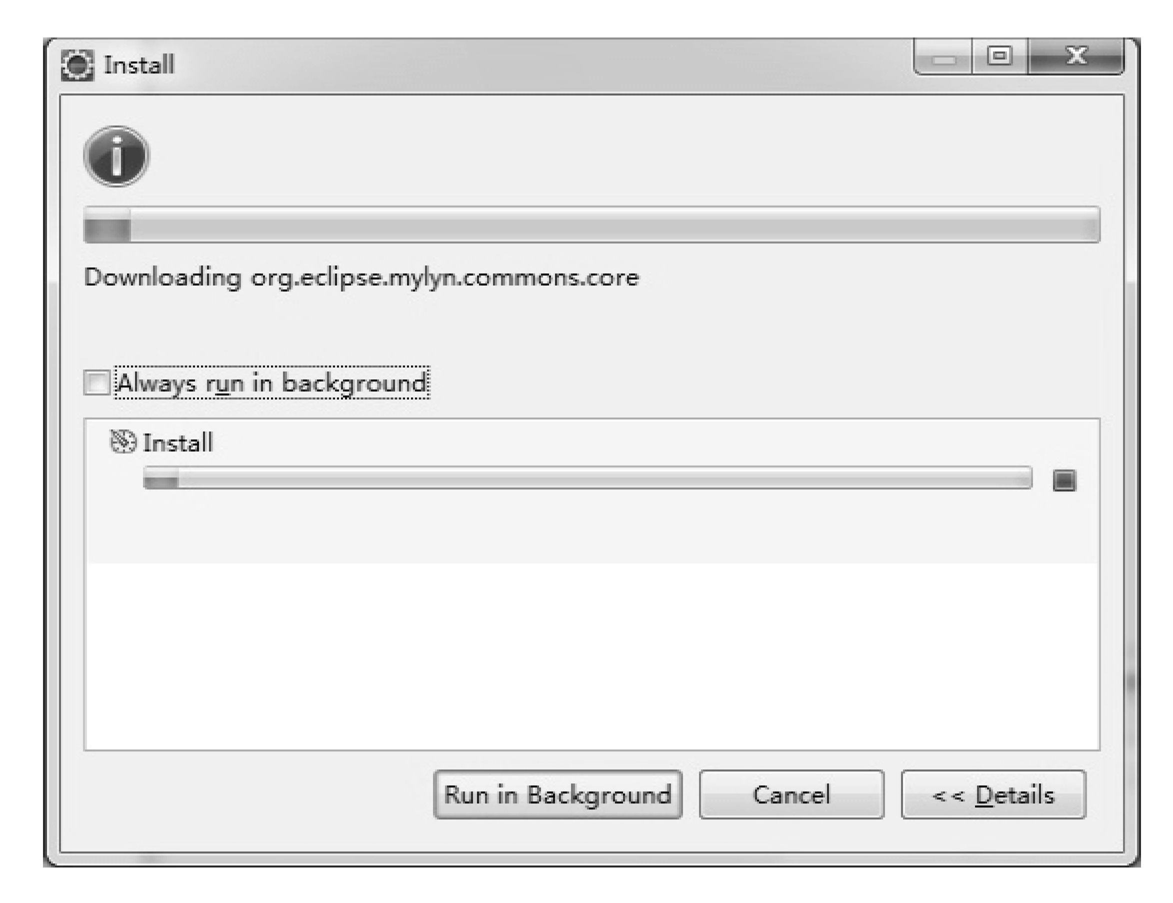


图 1-30　开始安装

第 6 步：当所有插件安装成功后，会弹出提示界面，如图 1-31 所示。

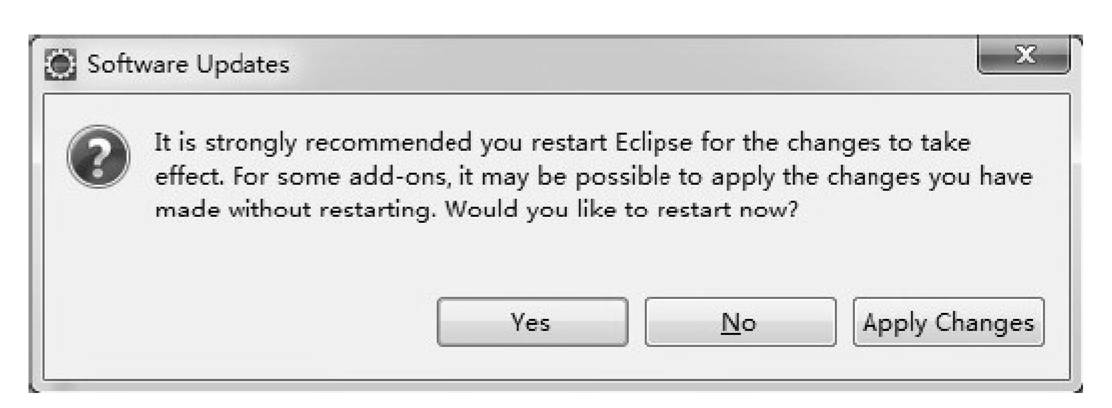


图 1-31　安装成功

这时需要单击 Yes 按钮重启 Eclipse 让所有插件生效。

（2）配置 Android SDK

打开 Eclipse，选择 Window | preferences 命令，进入如图 1-32 所示的界面。

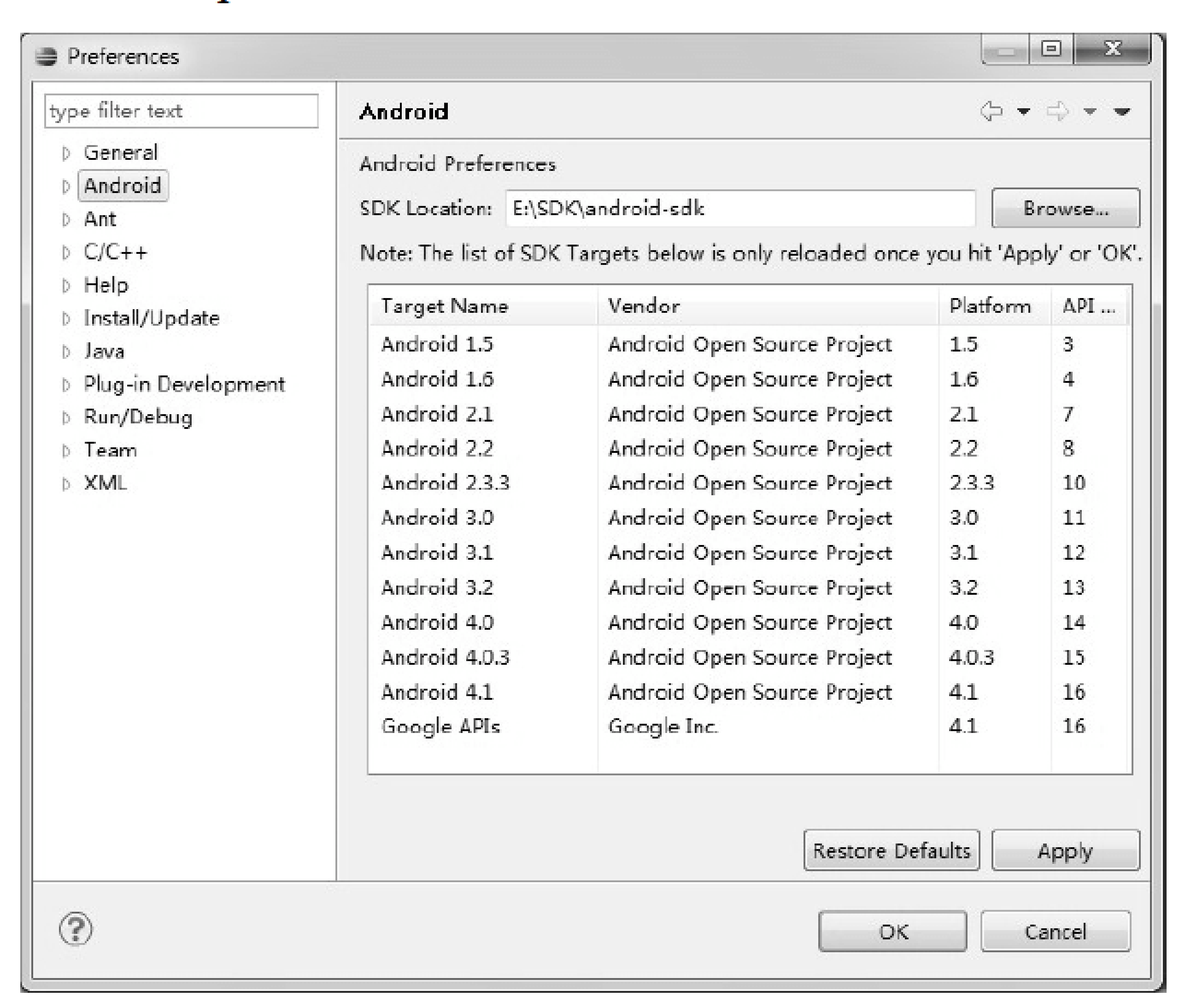


图 1-32　配置界面

这样就可以从 Eclipse 中新建 Android 工程，要想知道新建工程是基于什么版本的 Android 系统，可以打开 SDK 根目录下的 SDK 管理工具 SDK Manager.exe，双击后会进入到 SDK 工具包管理界面，如图 1-33 所示。

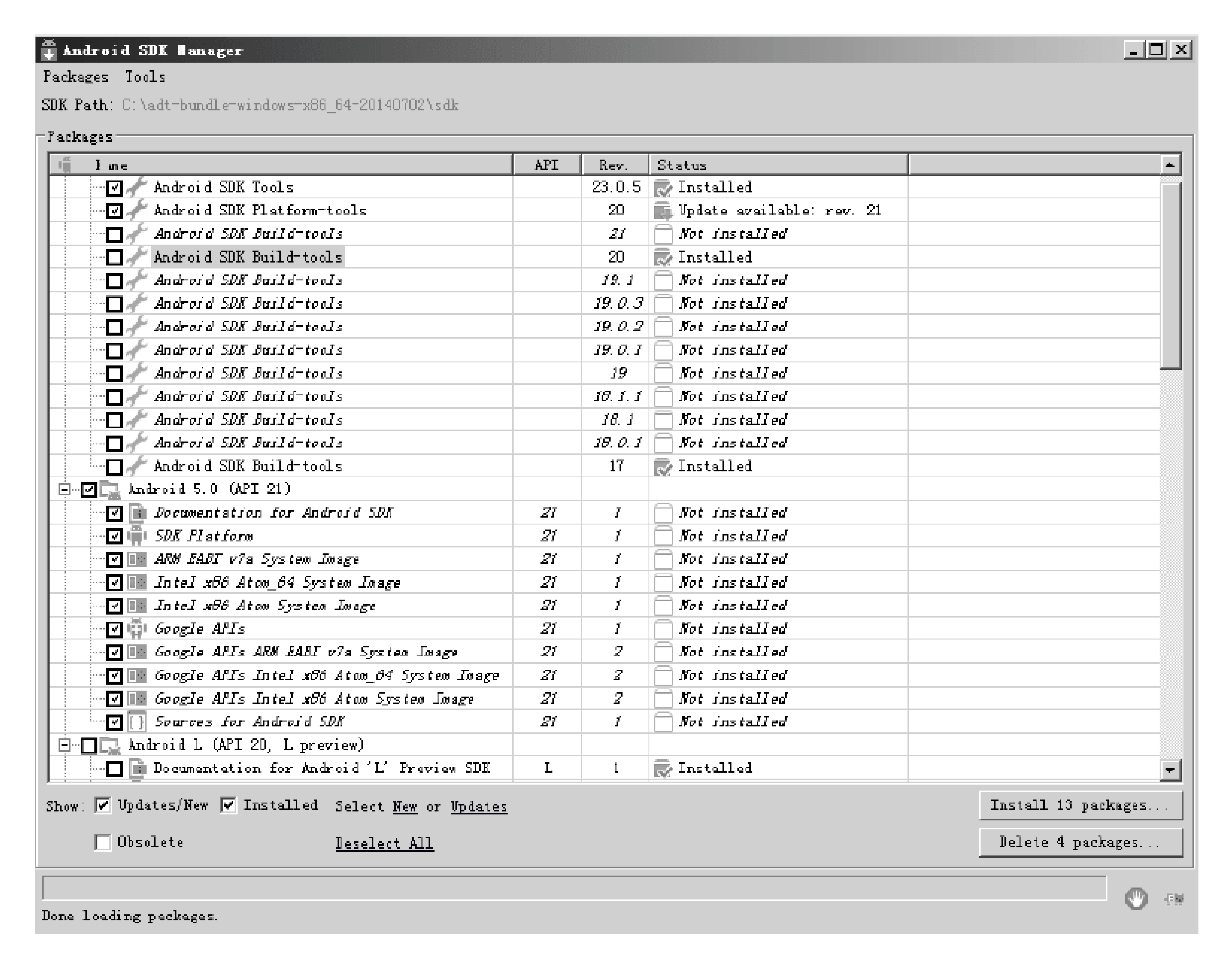


图 1-33　Android SDK 管理

从图 1-33 中可以看到很清晰地列出了当前版本 SDK 中包含的工具包，以及已经安装了的和没有安装的版本。可以继续单击 Install packages 或者 Delete packages 按钮安装和删除 SDK 中的工具包。如果是安装，则过程会比较慢，与网速的关系比较大。当将 SDK 中的工具包安装完毕，同时也完成了 Eclipse 和 SDK 的配置工作，至此 Windows 7 平台下基于 SDK 的 Android 的开发环境搭建全部完成。

第2章　分析 JNI

JNI 是 Java Native Interface 的缩写，表示 Java 本地接口。从 Java 1.1 开始，JNI 标准便成为了 Java 平台的一部分，它允许 Java 代码和其他语言写的代码进行交互。JNI 一开始是为了本地已编译语言，尤其是 C 和 C++而设计的，但这并不妨碍使用其他语言，只要调用约定受支持即可。本章将详细分析 Android 5.0 中 JNI 的基本知识。

2.1　JNI 基础

在 Android 系统中，JNI 是连接 Java 部分和 C/C++部分的桥梁。要想完整地使用 JNI，需要仔细分析 Java 代码和 C/C++代码。在 Android 中通过提供 JNI 的方式，让 Java 程序可以调用 C 语言程序。Android 中的很多 Java 类都具有 Native（本地）接口，这些接口由本地实现，然后注册到系统中。本节将详细讲解 JNI 的基础知识。

2.1.1　JNI 的功能结构

JNI 最初是由 Sun 提供的 Java 与系统中的原生方法交互的技术，用于在 Windows/Linux 系统中实现 Java 与 Native Method（本地方法）的相互调用。JVM（Java 虚拟机）在封装各种操作系统实际的差异性的同时提供了 JNI 技术，使得开发者可以通过 Java 程序（代码）调用到操作系统相关的技术实现的库函数，从而与其他技术和系统交互，使用其他技术实现系统的功能。同时，其他技术和系统也可以通过 JNI 提供的相应原生接口调用 Java 应用系统内部实现的功能。

在 Windows 系统上，一般可执行的应用程序都是基于 Native（本地）的 PE 结构，Windows 上的 JVM 也是基于 Native 结构实现的，Java 应用体系都是构建于 JVM 之上。由此可见，Windows 系统上的 Java 体系如图 2-1 所示。

JNI 对于应用本身来说，可以将其看作一个代理模式。对于开发者来说，需要使用 C/C++实现一个代理程序（JNI 程序）来实际操作目标原生函数，Java 程序中则是 JVM 通过加载并调用此 JNI 程序来间接地调用目标原生函数。JNI 的调用过程如图 2-2 所示。

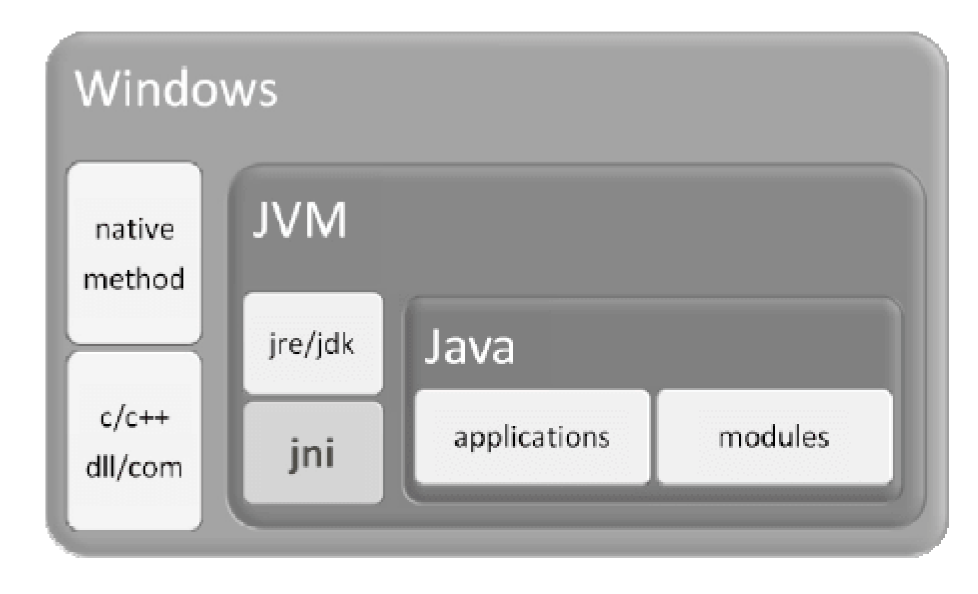


图 2-1　Windows 系统上的 Java 体系

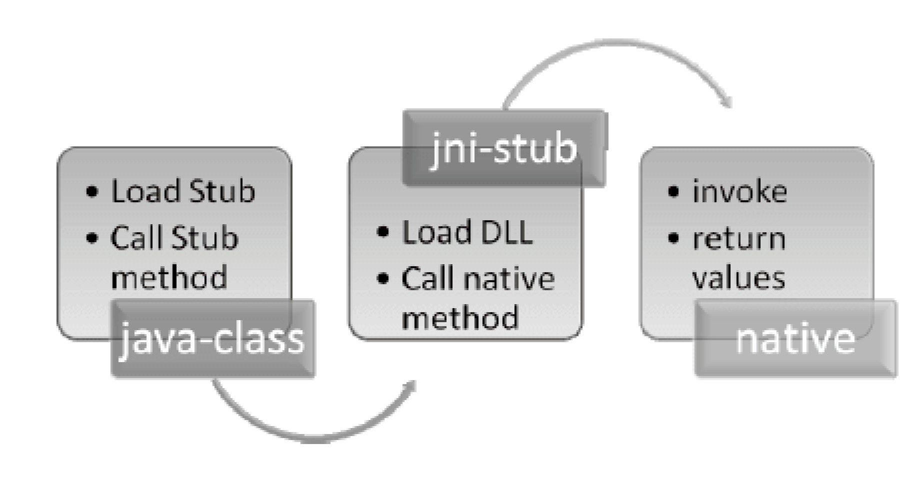


图 2-2　JNI 的调用过程图

2.1.2　JNI 的调用层次

JNI 调用的层次主要分为 3 层，在 Android 系统中这 3 层从上到下依次为 Java→JNI→C/C++（SO 库），Java 可以访问 C/C++中的方法，同样 C/C++可以修改 Java 对象，如图 2-3 所示为这三者之间的调用关系。

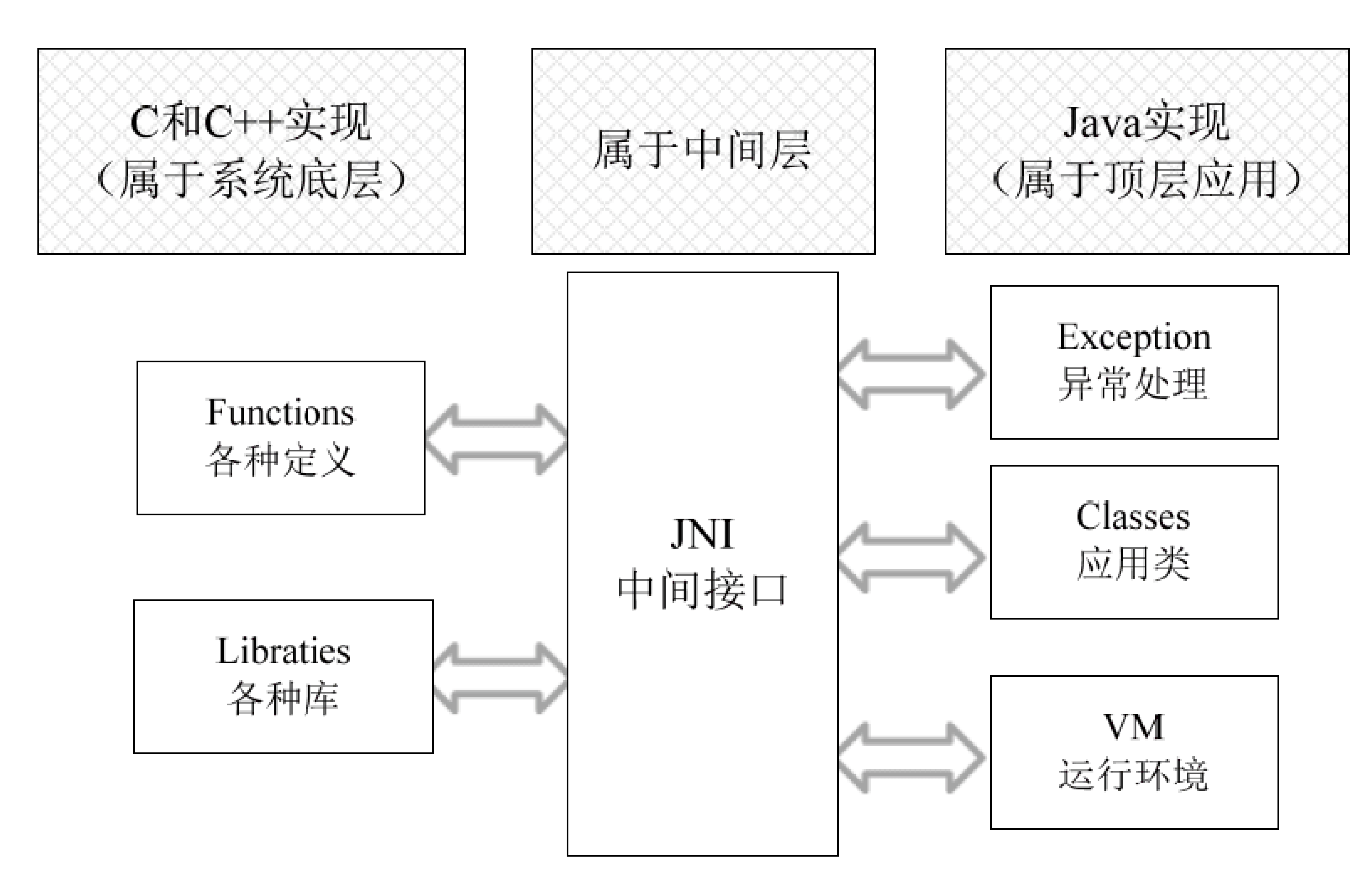


图 2-3　JNI 调用的层次关系

由图 2-3 可知，JNI 的调用关系为 Java→JNI→Native。

在 Android 5.0 的源码中，主要的 JNI 代码放在路径 frameworks/base/core/jni/中。

上述路径中的内容被编译成库 libandroid_runtime.so，这是一个普通的动态库，被放置在目标系统的/system/lib 目录下。另外，Android 中还存在其他的 JNI 库，其实 JNI 中的各个文件就是普通的 C++源文件；在 Android 中实现的 JNI 库，需要连接动态库 libnativehelper.so。

2.1.3　分析 JNI 的本质

要想弄明白 JNI 的本质，还要从 Java 的本质说起。从本质上来说，Java 语言的运行完全依赖于脚本引擎对 Java 的代码进行解释和执行。因为现代的 Java 可以从源代码编译成.class 之类的中间格式的二进制文件，所以这种处理会加快 Java 脚本的运行速度。尽管如此，基本的执行方式仍然不变，由脚本引擎（被称之为 JVM）来执行。与 python、perl 之类的纯脚本相比，只是把脚本变成了二进制格式而已。另外，Java 本身就是一门面向对象语言，可以调用完善的功能库。当把这个脚本引擎移植到所有平台上之后，这个脚本就很自然地实现“跨平台”了。绝大多数的脚本引擎都支持一个很显著的特性，就是可以通过 C/C++编写模块，并在脚本中调用这些模块。Java 也是如此，Java 一定要提供一种在脚本中调用 C/C++编写的模块的机制，才能称得上是一个完善的脚本引擎。

从本质上来看，Android 平台是由 arm-linux 操作系统和一个 Dalvik 虚拟机组成的。所有在 Android 模拟器上看到的界面效果都是用 Java 语言编写的，具体请看源代码中的 frameworks/base 目录。Dalvik 虚拟机只是提供了一个标准的支持 JNI 调用的 Java 虚拟机环境。

在 Android 平台中，使用 JNI 技术封装了所有和硬件相关的操作，通过 Java 去调用 JNI 模块，而 JNI 模块使用 C/C++调用 Android 本身的 arm-linux 底层驱动，这样便实现了对硬件的调用。

在 Android 5.0 的源码中，和 JNI 相关的文件如下所示。

☑ ./frameworks/base/media/java/android/media/MediaScanner.java

☑ ./frameworks/base/media/jni/android_media_MediaScanner.cpp

☑ ./frameworks/base/media/jni/android_media_MediaPlayer.cpp

☑ ./frameworks/base/media/jni/AndroidRuntime.cpp

☑ ./libnativehelper/JNIHelp.cpp

由此可见，和 JNI 密切相关的是 Media 系统，而 Media 系统的架构基础是 MediaScanner。在启动 Android 系统之初，就会扫描出系统中的 Media 文件供后续应用使用，既有新加入的媒体，也有几微秒钟前删除的媒体文件，并且还需要自动更新相应的媒体库。在 Android 系统中，和用户体验密切相关的 Music、Gallery 播放等应用，也是基于 MediaScanner 的扫描媒体文件功能的。MediaScanner 位于 Android 5.0 源码的路径 packages/providers/MediaProvider 中。

MediaProvider Manifest 包含了 3 个主要部分：MediaScannerReceiver、MediaScannerService 和 MediaProvider。在 MediaProvider 目录下的 AndroidManifest 中可以查看 MediaProvider 的基本架构，如图 2-4 所示。

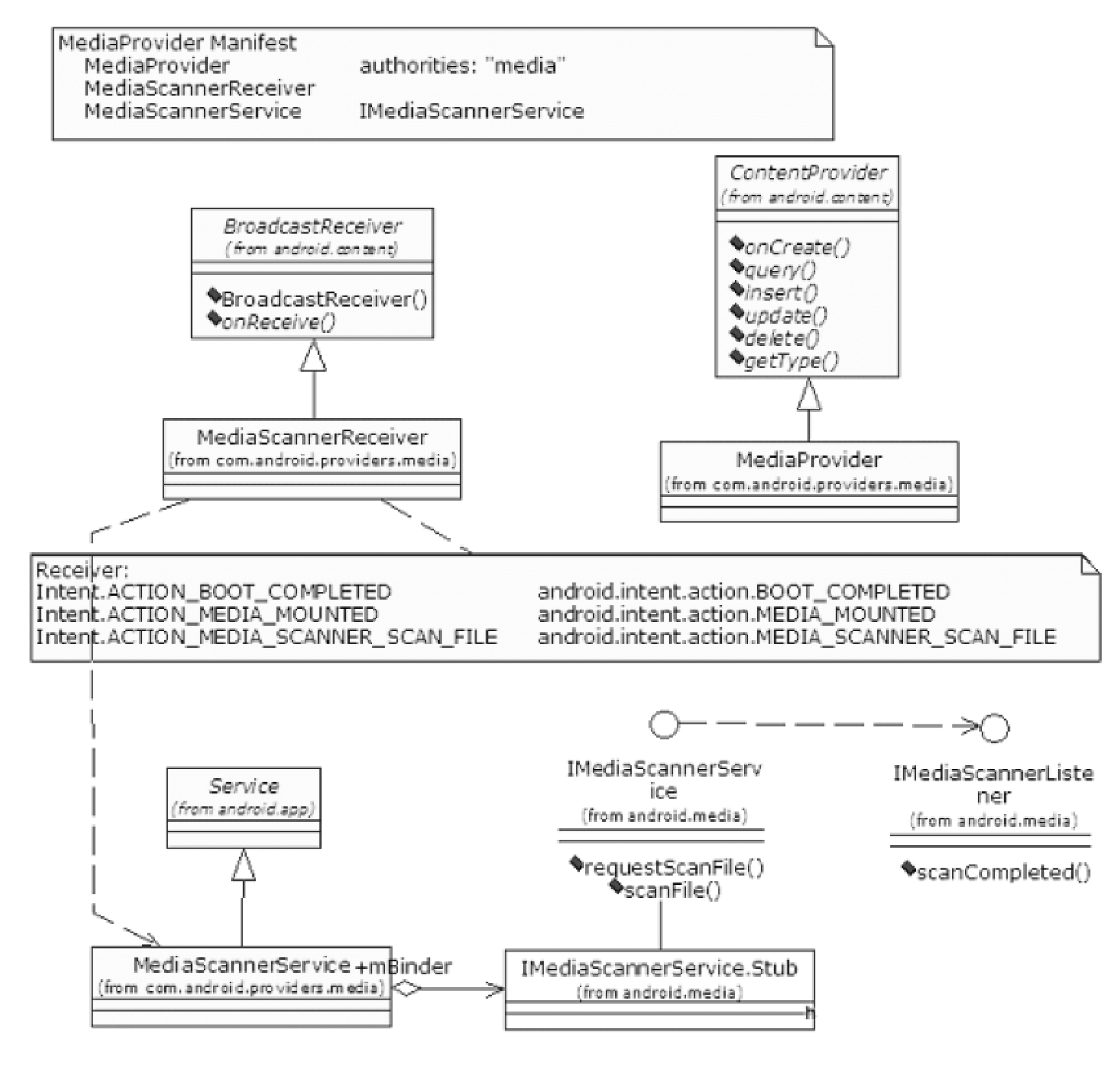


图 2-4　MediaProvider 的基本架构

☑ MediaScannerReceiver：是一个 BroadcastReceiver（接收广播），功能是进行媒体扫描，这也是

MediaScanner 提供给外界的接口之一。收到广播之后启动 MediaScannerService 执行扫描工作。

☑ MediaScannerService：是一个 Service，负责媒体扫描，还要用到 Framework 中的 MediaScanner 来共同完成具体扫描工作，扫描的结果在 MediaProvider 提供的数据库中。

☑ MediaProvider：是一个 ContentProvider，媒体库（Images/Audio/Video/Playlist 等）的数据提供者。负责操作数据库，并提供给其他的程序 insert、query、delete、update 等操作。

2.2 分析 MediaScanner

在 Android 5.0 中，下面是 MediaScanner 系统的 JNI 的调用关系。

```
MediaScanner -------------libmedia_jni.so -------------libmedia.so
```

在 Android 系统中，MediaScanner 的功能是扫描媒体文件，得到诸如歌曲时长、歌曲作者等信息，并将这些信息存放到媒体数据库中，以供其他应用程序使用。本节以 MediaScanner 源码分析作为基础，将详细分析 JNI 在 Android 系统中的作用。

2.2.1 分析 Java 层

在 MediaScanner 系统中，Java 层的实现文件为 frameworks/base/media/java/android/media/MediaScanner.java。

下面将详细讲解 MediaScanner 系统中 Java 层的具体实现源码。

（1）加载 JNI 库

在文件 MediaScanner.java 中，首先定义 MediaScanner 类并加载 JNI 库，然后定义 JNI 的 Native（本地）函数。主要代码如下所示。

```
public class MediaScanner
{
    static {
        System.loadLibrary("media_jni");
        native_init();
    }

    private final static String TAG = "MediaScanner";

    private static final String[] FILES_PRESCAN_PROJECTION = new String[] {
            Files.FileColumns._ID, // 0
            Files.FileColumns.DATA, // 1
            Files.FileColumns.FORMAT, // 2
            Files.FileColumns.DATE_MODIFIED, // 3
    };

    private static final String[] ID_PROJECTION = new String[] {
            Files.FileColumns._ID,
    };
```

```
    private static final int FILES_PRESCAN_ID_COLUMN_INDEX = 0;
    private static final int FILES_PRESCAN_PATH_COLUMN_INDEX = 1;
    private static final int FILES_PRESCAN_FORMAT_COLUMN_INDEX = 2;
    private static final int FILES_PRESCAN_DATE_MODIFIED_COLUMN_INDEX = 3;

    private static final String[] PLAYLIST_MEMBERS_PROJECTION = new String[] {
            Audio.Playlists.Members.PLAYLIST_ID, // 0
     };

    private static final int ID_PLAYLISTS_COLUMN_INDEX = 0;
    private static final int PATH_PLAYLISTS_COLUMN_INDEX = 1;
    private static final int DATE_MODIFIED_PLAYLISTS_COLUMN_INDEX = 2;

    private static final String RINGTONES_DIR = "/ringtones/";
    private static final String NOTIFICATIONS_DIR = "/notifications/";
    private static final String ALARMS_DIR = "/alarms/";
    private static final String MUSIC_DIR = "/music/";
    private static final String PODCAST_DIR = "/podcasts/";
…
    private static native final void native_init();//声明一个 native 函数，native 为关键字
    private native final void native_setup();
    …
}
```

函数 native_init()位于包 android.media 中，其完整路径名为 android.media.MediaScanner.nantive_init。根据规则，其对应的 JNI 层函数名称为 android_media_MediaScanner_native_init。

在调用 native 函数之前需要先加载 JNI 库，一般在类的 static 中加载调用函数 System.loadLibrary()。在加载了相应的 JNI 库之后，如果要使用相应的 native 函数，只需使用 native 声明需要被调用的函数即可。

```
private native void processDirectory(String path, String extensions, MediaScannerClient client);
private native void processFile(String path, String mimeType, MediaScannerClient client);
public native void setLocale(String locale);
```

（2）实现扫描工作

在文件 MediaScanner.java 中，通过函数 scanDirectories()实现扫描工作，具体代码如下所示。

```
public void scanDirectories(String[] directories, String volumeName) {
    try {
        long start = System.currentTimeMillis();
        initialize(volumeName);           //初始化
        prescan(null, true);              //扫描前的预处理
        long prescan = System.currentTimeMillis();

        if (ENABLE_BULK_INSERTS) {
            mMediaInserter = new MediaInserter(mMediaProvider, 500);
        }
        //函数 processDirectory()是一个 Native 函数，功能是对目标文件夹进行扫描
        for (int i = 0; i < directories.length; i++) {
```

```
                processDirectory(directories[i], mClient);
            }

            if (ENABLE_BULK_INSERTS) {
                mMediaInserter.flushAll();
                mMediaInserter = null;
            }

            long scan = System.currentTimeMillis();
            postscan(directories);//扫描后的处理
            long end = System.currentTimeMillis();

            if (false) {
                Log.d(TAG, " prescan time: " + (prescan - start) + "ms\n");
                Log.d(TAG, " scan time: " + (scan - prescan) + "ms\n");
                Log.d(TAG, "postscan time: " + (end - scan) + "ms\n");
                Log.d(TAG, " total time: " + (end - start) + "ms\n");
            }
        } catch (SQLException e) {
            Log.e(TAG, "SQLException in MediaScanner.scan()", e);
        } catch (UnsupportedOperationException e) {
            Log.e(TAG, "UnsupportedOperationException in MediaScanner.scan()", e);
        } catch (RemoteException e) {
            Log.e(TAG, "RemoteException in MediaScanner.scan()", e);
        }
}
```

在上述代码中使用函数 initialize()实现初始化操作，此函数的具体实现代码如下所示。

```
private void initialize(String volumeName) {
    //打开 MediaProvider，获得它的一个实例
    mMediaProvider = mContext.getContentResolver().acquireProvider("media");
    //得到一些 Uri
    mAudioUri = Audio.Media.getContentUri(volumeName);
    mVideoUri = Video.Media.getContentUri(volumeName);
    mImagesUri = Images.Media.getContentUri(volumeName);
    mThumbsUri = Images.Thumbnails.getContentUri(volumeName);
    mFilesUri = Files.getContentUri(volumeName);
    //如果需要外部存储，则可以支持播放列表，用缓存池实现，例如 mGenerCache 等
    if (!volumeName.equals("internal")) {
        mProcessPlaylists = true;
        mProcessGenres = true;
        mPlaylistsUri = Playlists.getContentUri(volumeName);
        mCaseInsensitivePaths = true;
    }
}
```

（3）读取并保存信息

在文件 MediaScanner.java 中，通过函数 prescan()读取之前扫描的数据库中和文件相关的信息并保存起来。函数 prescan()创建了一个用于缓存扫描文件信息的对象 FileCache，例如 last_modified 等。这

个 FileCache 是从 MediaProvider 中已有信息构建出来的历史信息，并且根据扫描得到的新信息来对应更新历史信息。函数 prescan()的具体实现代码如下所示。

```
private void prescan(String filePath, boolean prescanFiles) throws RemoteException {
    Cursor c = null;
    String where = null;
    String[] selectionArgs = null;
    //mPlayLists 保存从数据库中获取的信息
    if (mPlayLists == null) {
        mPlayLists = new ArrayList<FileEntry>();
    } else {
        mPlayLists.clear();
    }

    if (filePath != null) {
        //只有一个文件查询
        where = MediaStore.Files.FileColumns._ID + ">?" +
            " AND " + Files.FileColumns.DATA + "=?";
        selectionArgs = new String[] { "", filePath };
    } else {
        where = MediaStore.Files.FileColumns._ID + ">?";
        selectionArgs = new String[] { "" };
    }

    //告诉提供者不删除文件，如果不需要删除文件，则需要避免意外删除这个文件的机制
    //可能在系统未被安装和未安装在扫描仪之前发生
    Uri.Builder builder = mFilesUri.buildUpon();
    builder.appendQueryParameter(MediaStore.PARAM_DELETE_DATA, "false");
    MediaBulkDeleter deleter = new MediaBulkDeleter(mMediaProvider, builder.build());

    //根据内容提供者建立文件列表
    try {
        if (prescanFiles) {
            //首先从文件表读到现有文件
            //因为可能存在删除不存在文件的情况，所以要小批量地实现数据库查询以避免这个问题
            long lastId = Long.MIN_VALUE;
            Uri limitUri = mFilesUri.buildUpon().appendQueryParameter("limit", "1000").build();
            mWasEmptyPriorToScan = true;

            while (true) {
                selectionArgs[0] = "" + lastId;
                if (c != null) {
                    c.close();
                    c = null;
                }
                c = mMediaProvider.query(limitUri, FILES_PRESCAN_PROJECTION,
                        where, selectionArgs, MediaStore.Files.FileColumns._ID, null);
                if (c == null) {
                    break;
                }
```

```
                    int num = c.getCount();

                    if (num == 0) {
                        break;
                    }
                    mWasEmptyPriorToScan = false;
                    while (c.moveToNext()) {
                        long rowId = c.getLong(FILES_PRESCAN_ID_COLUMN_INDEX);
                        String path = c.getString(FILES_PRESCAN_PATH_COLUMN_INDEX);
                        int format = c.getInt(FILES_PRESCAN_FORMAT_COLUMN_INDEX);
                        long lastModified = c.getLong(FILES_PRESCAN_DATE_MODIFIED_COLUMN_INDEX);
                        lastId = rowId;
                        if (path != null && path.startsWith("/")) {
                            boolean exists = false;
                            try {
                                exists = Libcore.os.access(path, libcore.io.OsConstants.F_OK);
                            } catch (ErrnoException e1) {
                            }
                            if (!exists && !MtpConstants.isAbstractObject(format)) {
                                MediaFile.MediaFileType mediaFileType = MediaFile.getFileType(path);
                                int fileType = (mediaFileType == null ? 0 : mediaFileType.fileType);

                                if (!MediaFile.isPlayListFileType(fileType)) {
                                    deleter.delete(rowId);
                                    if (path.toLowerCase(Locale.US).endsWith("/.nomedia")) {
                                        deleter.flush();
                                        String parent = new File(path).getParent();
                                        mMediaProvider.call(MediaStore.UNHIDE_CALL, parent, null);
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    finally {
        if (c != null) {
            c.close();
        }
        deleter.flush();
    }

    //计算图像的原始尺寸
    mOriginalCount = 0;
    c = mMediaProvider.query(mImagesUri, ID_PROJECTION, null, null, null, null);
    if (c != null) {
        mOriginalCount = c.getCount();
        c.close();
    }
}
```

（4）删除不存在于 SD 卡中的文件信息

在文件 MediaScanner.java 中，函数 postscan()的功能是删除不存在于 SD 卡中的文件信息，具体实现代码如下所示。

```
private void postscan(String[] directories) throws RemoteException {

    //触发播放列表后能够知道对应存储的媒体文件
    if (mProcessPlaylists) {
        processPlayLists();
    }

    if (mOriginalCount == 0 && mImagesUri.equals(Images.Media.getContentUri("external")))
        pruneDeadThumbnailFiles();

    //允许 GC 清理
    mPlayLists = null;
    mMediaProvider = null;
}
```

（5）processDirectory

在文件 MediaScanner.java 中，本地方法 processDirectory()能够直接转向 JNI，具体实现代码如下所示。

```
static void android_media_MediaScanner_processDirectory(JNIEnv *env, jobject thiz, jstring path, jstring
extensions, jobject client)
{    //获取 MediaScanner
     MediaScanner *mp = (MediaScanner *)env->GetIntField(thiz, fields.context);
     //参数判断，并抛出异常
     if (path == NULL) {
         jniThrowException(env, "java/lang/IllegalArgumentException", NULL);
         return;
     }
     if (extensions == NULL) {
         jniThrowException(env, "java/lang/IllegalArgumentException", NULL);
         return;
     }

     const char *pathStr = env->GetStringUTFChars(path, NULL);
     if (pathStr == NULL) {
         jniThrowException(env, "java/lang/RuntimeException", "Out of memory");
         return;
     }
     const char *extensionsStr = env->GetStringUTFChars(extensions, NULL);
     if (extensionsStr == NULL) {
         env->ReleaseStringUTFChars(path, pathStr);
         jniThrowException(env, "java/lang/RuntimeException", "Out of memory");
         return;
     }
     //初始化 client 实例
     MyMediaScannerClient myClient(env, client);
     //mp 调用 processDirectory()
```

```
        mp->processDirectory(pathStr, extensionsStr, myClient, ExceptionCheck, env);
        //gc
        env->ReleaseStringUTFChars(path, pathStr);
        env->ReleaseStringUTFChars(extensions, extensionsStr);
}
```

（6）扫描函数 scanFile()

函数 scanFile()的功能是调用函数 doScanFile()对指定的文件进行扫描，具体实现代码如下所示。

```
public void scanFile(String path, long lastModified, long fileSize,
        boolean isDirectory, boolean noMedia) {
    //这是来自本地代码的回调函数
    // Log.v(TAG, "scanFile: "+path);
    doScanFile(path, null, lastModified, fileSize, isDirectory, false, noMedia);
}
```

（7）异常处理

在 Android 5.0 中，为了处理 Java 实现的方法中和 C/C++实现方法中抛出的 Java 异常，JNI 特意提供了一套异常处理机制函数集，以专门用于检查、分析和处理异常情况。例如，在文件 jni.h 中定义了主要的异常函数，具体代码如下所示。

```
//抛出异常
jint (*Throw)(JNIEnv*, jthrowable);
//抛出新的异常
jint (*ThrowNew)(JNIEnv *, jclass, const char *);
//异常产生
jthrowable (*ExceptionOccurred)(JNIEnv*);
void (*ExceptionDescribe)(JNIEnv*);
//清除异常
void (*ExceptionClear)(JNIEnv*);
void (*FatalError)(JNIEnv*, const char*);
```

例如，在 Camera 模块中也用到了异常处理，在文件 android_hardware_Camera.cpp 中也涉及了异常操作，具体代码实例如下所示。

```
void JNICameraContext::copyAndPost(JNIEnv* env, const sp<IMemory>& dataPtr, int msgType)
{
  …
  if (obj == NULL) {
              LOGE("Couldn't allocate byte array for JPEG data");
              env->ExceptionClear();
          } else {
              env->SetByteArrayRegion(obj, 0, size, data);
          }
      } else {
          LOGE("image heap is NULL");
      }
    }
…
}
```

在文件 android_hardware_Camera.cpp 中，函数 android_hardware_Camera_startPreview()同样也用到了异常处理机制，具体代码如下所示。

```
static void android_hardware_Camera_startPreview(JNIEnv *env, jobject thiz)
{
    LOGV("startPreview");
    sp<Camera> camera = get_native_camera(env, thiz, NULL);
    if (camera == 0) return;
    if (camera->startPreview() != NO_ERROR) {
        jniThrowRuntimeException(env, "startPreview failed");
        return;
    }
}
```

在上述代码中，android_hardware_Camera_startPreview()如果发现 startPreview()函数返回错误，则会抛出异常并返回。这里的异常与 Java 中的异常机制很相似，读者可以对比分析其原理。

2.2.2　分析 JNI 层

由于 Android 应用层的类都是以 Java 写的，当这些 Java 类被编译为 Dex 型的 Bytecode（位元码，是一个程序处理的计算机目标代码，通常是指虚拟机，而不是真的计算机或硬件处理器）后，必须借助 Dalvik 虚拟机来执行并实现。虚拟机在 Android 系统中扮演了一个很重要的角色，并且在执行 Java 类的过程中，如果 Java 类需要与 C 组件沟通时，VM 就会载入 C 组件，然后让 Java 的函数顺利地调用到 C 组件的函数。此时，VM 扮演着桥梁的角色，让 Java 与 C 组件能通过标准的 JNI 界面相互沟通。

应用层的 Java 类是在虚拟机上执行的，而 C 组件不在 Android 虚拟机上执行。如果 Java 程序要求 Android 虚拟机载入（Load）所指定的 C 组件，可以使用如下指令实现这一功能。

```
System.loadLibrary(*.so 的档案名);
```

例如，在 Android 5.0 的框架中，文件 MediaPlayer.java 包含了如下指令。

```
public class MediaPlayer{
        static {
                System.loadLibrary("media_jni");
            }
    }
```

这要求 Android 虚拟机去载入 Android 的/system/lib/libmedia_jni.so 库。载入*.so 后，Java 类与*.so 档案就汇合起来一起执行。

在 JNI 层中，MediaScanner 的对应文件是./frameworks/base/media/jni/android_media_MediaScanner.cpp。下面将详细讲解 MediaScanner 系统 JNI 层的基本源码。

（1）将指针保存到 Java 对象

在文件 android_media_MediaScanner.cpp 中，函数 android_media_MediaScanner_native_init()的功能是将 Native 对象的指针保存到 Java 对象中。函数 android_media_MediaScanner_native_init()的具体实现代码如下所示。

```
static const char* const kClassMediaScanner =
            "android/media/MediaScanner";
       …

/*native_init()函数的 JNI 层实现*/
static void android_media_MediaScanner_native_init(JNIEnv *env)
{
      ALOGV("native_init");
      jclass clazz = env->FindClass(kClassMediaScanner);
      if (clazz == NULL) {
            return;
      }
      fields.context = env->GetFieldID(clazz, "mNativeContext", "I");
      if (fields.context == NULL) {
            return;
      }
}
```

（2）创建 Native 层的 MediaScanner 对象

在文件 android_media_MediaScanner.cpp 中，函数 android_media_MediaScanner_native_setup()的功能是创建一个 Native 层的 MediaScanner 对象，但是此函数使用的是 Opencore 提供的 PVMediaScanner。函数 android_media_MediaScanner_native_setup()的具体实现代码如下所示。

```
static void android_media_MediaScanner_native_setup(JNIEnv *env, jobject thiz)
{
      ALOGV("native_setup");
      MediaScanner *mp = new StagefrightMediaScanner;
      if (mp == NULL) {
            jniThrowException(env, kRunTimeException, "Out of memory");
            return;
      }
      env->SetIntField(thiz, fields.context, (int)mp);
}
```

2.2.3 分析 Native（本地）层

在现实应用中，Java 的 Native 函数与 JNI 函数是一一对应的关系。在 Android 5.0 中，使用 JNI NativeMethod 的结构体来记录这种对应关系。下面将详细分析 Mediascanner 系统中的 Native 层的实现源码。

（1）注册 JNI 函数

在 Android 系统中，使用了一种“特定”的方式来定义其 Native 函数，这与传统定义 Java JNI 的方式有所差别。其中很重要的区别是在 Android 中使用了一种 Java 和 C 函数的映射表数组，并在其中描述了函数的参数和返回值。这个数组的类型是 JNINativeMethod，具体定义如下所示。

```
typedef struct {
      const char* name;              /*Java 中函数的名字*/
      const char* signature;         /*描述了函数的参数和返回值*/
```

```
    void* fnPtr;                    /*函数指针，指向 C 函数*/
} JNINativeMethod;
```

在上述代码中，比较难以理解的是第二个参数，例如：

```
"()V"
"(II)V"
"(Ljava/lang/String;Ljava/lang/String;)V"
```

实际上这些字符是与函数的参数类型一一对应的，具体说明如下所示。

☑ ()中的字符表示参数，后面的则代表返回值。例如，"()V"就表示 void Func();。

☑ (II)V 表示 void Func(int, int);。

具体的每一个字符的对应关系如表 2-1 所示。

表 2-1　字符对应关系

字　符	Java 类型	C 类型
V	void	void
Z	jboolean	boolean
I	jint	int
J	jlong	long
D	jdouble	double
F	jfloat	float
B	jbyte	byte
C	jchar	char
S	jshort	short

而数组则以“[”开始，用两个字符表示，例如“[]”“[F”“[B”等，具体类型对应关系可参考 2.4 节。

上面的都是基本类型，如果 Java 函数的参数是 class，则以 L 开头，以“;”结尾，中间部分是用“/”隔开的包及类名。而其对应的 C 函数名的参数则为 jobject。一个例外是 String 类，其对应的类为 jstring，即：

☑ Ljava/lang/String 中的 String jstring。

☑ Ljava/net/Socket 中的 Socket jobject。

如果 Java 函数位于一个嵌入类，则使用“$”作为类名间的分隔符。例如：

```
(Ljava/lang/String$Landroid/os/FileUtils$FileStatus;)
```

定义并注册 JNINativeMethod 数组，对应的实现代码如下所示。

```
/*定义一个 JNINativeMethod 数组*/
static JNINativeMethod gMethods[] = {
    {
        "processDirectory",
        "(Ljava/lang/String;Landroid/media/MediaScannerClient;)V",
        (void *)android_media_MediaScanner_processDirectory
    },

    {
```

```
            "processFile",
            "(Ljava/lang/String;Ljava/lang/String;Landroid/media/MediaScannerClient;)V",
            (void *)android_media_MediaScanner_processFile
    },

    {
            "setLocale",
            "(Ljava/lang/String;)V",
            (void *)android_media_MediaScanner_setLocale
    },

    {
            "extractAlbumArt",
            "(Ljava/io/FileDescriptor;)[B",
            (void *)android_media_MediaScanner_extractAlbumArt
    },

    {
            "native_init",
            "()V",
            (void *)android_media_MediaScanner_native_init
    },

    {
            "native_setup",
            "()V",
            (void *)android_media_MediaScanner_native_setup
    },

    {
            "native_finalize",
            "()V",
            (void *)android_media_MediaScanner_native_finalize
    },
};
/*注册 JNINativeMethod 数组*/
int register_android_media_MediaScanner(JNIEnv *env)
{
    return AndroidRuntime::registerNativeMethods(env,
                kClassMediaScanner, gMethods, NELEM(gMethods));
}
```

（2）实现注册工作

定义并注册数组 JNINativeMethod 后，接着需要在文件 AndroidRuntime.cpp 中调用函数 registerNativeMethods()来完成调用工作，具体实现代码如下所示。

```
int AndroidRuntime::registerNativeMethods(JNIEnv* env,
    const char* className, const JNINativeMethod* gMethods, int numMethods)
{
    return jniRegisterNativeMethods(env, className, gMethods, numMethods);
}
```

在上述代码中，函数 jniRegisterNativeMethods()在文件 JNIHelp.cpp 中实现，这是 Android 为方便 JNI 使用而提供的一个帮助函数，具体实现代码如下所示。

```
extern "C" int jniRegisterNativeMethods(C_JNIEnv* env, const char* className,
    const JNINativeMethod* gMethods, int numMethods)
{
    JNIEnv* e = reinterpret_cast<JNIEnv*>(env);

    ALOGV("Registering %s natives", className);
    scoped_local_ref<jclass> c(env, findClass(env, className));
    if (c.get() == NULL) {
        ALOGE("Native registration unable to find class '%s', aborting", className);
        abort();
    }
    if ((*env)->RegisterNatives(e, c.get(), gMethods, numMethods) < 0) {
        ALOGE("RegisterNatives failed for '%s', aborting", className);
        abort();
    }
    return 0;
}
```

通过上述代码可以了解函数 registerNativeMethods()的作用。应用层级的 Java 类别穿过 Android 虚拟机呼叫到本地函数，这个过程通常是通过 Android 虚拟机去寻找*.so 格式库文件中的本地函数。如果需要连续呼叫很多次，则需要每次都寻找一遍，这会多花费很多时间。此时，组件开发人员可以自行向 Android 虚拟机登记本地函数。例如，在 Android 的/system/lib/libmedia_jni.so 档案中的代码片段如下所示：

```
#define LOG_TAG "MediaPlayer-JNI"
static JNINativeMethod gMethods[] = {
    {"setDataSource", "(Ljava/lang/String;)V",
     (void *)android_media_MediaPlayer_setDataSource},
    {"setDataSource", "(Ljava/io/FileDescriptor;JJ)V",
(void *)android_media_MediaPlayer_setDataSourceFD},

    {"prepare", "()V", (void *)android_media_MediaPlayer_prepare},
    {"prepareAsync", "()V", (void *)android_media_MediaPlayer_prepareAsync},
    {"_start", "()V", (void *)android_media_MediaPlayer_start},
    {"_stop", "()V", (void *)android_media_MediaPlayer_stop},
    {"getVideoWidth", "()I", (void *)android_media_MediaPlayer_getVideoWidth},
    {"getVideoHeight", "()I", (void *)android_media_MediaPlayer_getVideoHeight},
    {"seekTo", "(I)V", (void *)android_media_MediaPlayer_seekTo},
    {"_pause", "()V", (void *)android_media_MediaPlayer_pause},
    {"isPlaying", "()Z", (void *)android_media_MediaPlayer_isPlaying},
    {"getCurrentPosition", "()I", (void *)android_media_MediaPlayer_getCurrentPosition},
    {"getDuration", "()I", (void *)android_media_MediaPlayer_getDuration},
    {"_release", "()V", (void *)android_media_MediaPlayer_release},
    {"_reset", "()V", (void *)android_media_MediaPlayer_reset},
    {"setAudioStreamType","(I)V", (void *)android_media_MediaPlayer_setAudioStreamType},
    {"setLooping", "(Z)V", (void *)android_media_MediaPlayer_setLooping},
    {"setVolume", "(FF)V", (void *)android_media_MediaPlayer_setVolume},
```

```
    {"getFrameAt", "(I)Landroid/graphics/Bitmap;",
            (void *)android_media_MediaPlayer_getFrameAt},
    {"native_setup", "(Ljava/lang/Object;)V",
              (void *)android_media_MediaPlayer_native_setup},
    {"native_finalize", "()V", (void *)android_media_MediaPlayer_native_finalize},
};
static int register_android_media_MediaPlayer(JNIEnv *env){
    return AndroidRuntime::registerNativeMethods(env,
                "android/media/MediaPlayer", gMethods, NELEM(gMethods));
}
jint JNI_OnLoad(JavaVM* vm, void* reserved){
    if (register_android_media_MediaPlayer(env) < 0) {
        LOGE("ERROR: MediaPlayer native registration failed\n");
        goto bail;
    }
}
```

这样当 Android 虚拟机载入 libmedia_jni.so 档案时，就会呼叫函数 JNI_OnLoad()，然后 JNI_OnLoad() 呼叫函数 register_android_media_MediaPlayer()。此时，就呼叫函数 AndroidRuntime::registerNativeMethods()，并向 Android 虚拟机（即 AndroidRuntime）登记数组 gMethods[]表格所含的本地函数。由此可见，函数 registerNativeMethods()具备如下两个功能。

☑ 更有效率地找到函数。

☑ 可以在执行期间进行抽换。因为 gMethods[]是一个“<名称,函数指针>”格式的对照表，所以在执行程序时，可以通过多次呼叫函数 registerNativeMethods()的方式来更换本地函数的指针。

（3）实现动态注册

当 Java 层通过 System.loadLibrary 加载完 JNI 动态库后，会查找函数 JNI_OnLoad()，通过调用文件 android_media_MediaPlayer.cpp 中的函数 JNI_OnLoad()来完成动态注册工作，具体实现代码如下所示。

```
jint JNI_OnLoad(JavaVM* vm, void* reserved)
{
    JNIEnv* env = NULL;
    jint result = -1;

    if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        ALOGE("ERROR: GetEnv failed\n");
        goto bail;
    }
    assert(env != NULL);
    …
    if (register_android_media_MediaScanner(env) < 0) {
        ALOGE("ERROR: MediaScanner native registration failed\n");
        goto bail;
    }
    …
    /*成功，则返回有效的版本号*/
    result = JNI_VERSION_1_4;
bail:
    return result;
}
```

在上述代码中，函数 JNI_OnLoad()会回传 JNI_VERSION_1_4 的值给 Android 虚拟机，这样 Android 虚拟机能够知道所使用 JNI 的版本。此外，它也做了一些初期的动作（可呼叫任何本地函数），例如下面的指令：

```
if (register_android_media_MediaPlayer(env) < 0) {
        LOGE("ERROR: MediaPlayer native registration failed\n");
        goto bail;
    }
```

这样就将此组件提供的各个本地函数（Native Function）登记到 Android 虚拟机中，以便能加快后续呼叫本地函数的效率。

函数 JNI_OnUnload()与 JNI_OnLoad()是相对应的。在载入 C 组件时会立即呼叫 JNI_OnLoad()进行组件内的初期动作。当 Android 虚拟机释放该 C 组件时，则会呼叫 JNI_OnUnload()函数来进行善后清除动作。当 VM 呼叫 JNI_OnLoad()或 JNI_Unload()函数时，都会将 Android 虚拟机的指针（Pointer）传递给它们，其具体参数如下所示。

```
jint JNI_OnLoad(JavaVM* vm, void* reserved) {}
jint JNI_OnUnload(JavaVM* vm, void* reserved){}
```

在 JNI_OnLoad()函数中，通过 Android 虚拟机的指标取得 JNIEnv 的指标，并存入 env 指标变量中，如下述指令所示。

```
jint JNI_OnLoad(JavaVM* vm, void* reserved){
     JNIEnv* env = NULL;
     jint result = -1;
    if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
          LOGE("ERROR: GetEnv failed\n");
          goto bail;
    }
}
```

因为 Android 虚拟机通常是多执行绪（Multi-threading）的执行环境。每一个执行绪在呼叫 JNI_OnLoad()时，传递进来的 JNIEnv 指标值都是不同的。为了配合这种多执行绪的环境，C 组件开发者在撰写本地函数时，可借用由 JNIEnv 指标值的不同而避免执行绪的资料冲突问题，才能确保所写的本地函数能安全地在 Android 虚拟机中执行。基于这个原因，在呼叫 C 组件的函数时会将 JNIEnv 指标值传递给它，对应的实现代码如下所示。

```
jint JNI_OnLoad(JavaVM* vm, void* reserved)
{
JNIEnv* env = NULL;
    if (register_android_media_MediaPlayer(env) < 0) {
   }
}
```

这样当 JNI_OnLoad()呼叫函数 register_android_media_MediaPlayer(env)时，就将 env 指标值传递过去。这样函数 register_android_media_MediaPlayer()就能借用该标识值来区别不同的执行，以便解决资料冲突的问题。

例如，在 register_android_media_MediaPlayer()函数中可以编写如下指令。

```
if ((*env)->MonitorEnter(env, obj) != JNI_OK) {
}
```

此时可以查看是否有其他执行程序进入此物件，如果没有，则此执行就进入该物件中开始执行，并且也可以编写如下指令。

```
if ((*env)->MonitorExit(env, obj) != JNI_OK) {
}
```

这样便可以查看是否此执行正在此物件内执行，如果是，此执行就会立即离开。

（4）处理路径参数

在文件 frameworks/base/media/libmedia/MediaScanner.cpp 中，函数 processDirectory()的功能是调用 doProcessDirectory()处理路径参数。其参数@extensions 可能包含多个扩展名，在扩展名之间用“,”分隔开。函数 processDirectory()的具体实现代码如下所示。

```
status_t MediaScanner::processDirectory(
        const char *path, const char *extensions,
        MediaScannerClient &client,
        ExceptionCheck exceptionCheck, void *exceptionEnv) {
    int pathLength = strlen(path);
    if (pathLength >= PATH_MAX) {
        return UNKNOWN_ERROR;
    }
    char* pathBuffer = (char *)malloc(PATH_MAX + 1);
    if (!pathBuffer) {
        return UNKNOWN_ERROR;
    }

    int pathRemaining = PATH_MAX - pathLength;
    strcpy(pathBuffer, path);
    if (pathLength > 0 && pathBuffer[pathLength - 1] != '/') {
        pathBuffer[pathLength] = '/';
        pathBuffer[pathLength + 1] = 0;
        --pathRemaining;
    }

    client.setLocale(locale());

    status_t result =
        doProcessDirectory(
                pathBuffer, pathRemaining, extensions, client,
                exceptionCheck, exceptionEnv);

    free(pathBuffer);

    return result;
}
```

（5）扫描文件

当收到扫描某个文件的请求时，会调用函数 scanFile()来扫描这个文件。函数 scanFile()的具体实现代码如下所示。

```
virtual bool scanFile(const char* path, long long lastModified, long long fileSize)
  {
        jstring pathStr;
        if ((pathStr = mEnv->NewStringUTF(path)) == NULL) return false;
        //调用 Java 里 mClient 中的 scanFile()方法
        mEnv->CallVoidMethod(mClient, mScanFileMethodID, pathStr, lastModified, fileSize);
        mEnv->DeleteLocalRef(pathStr);
        return (!mEnv->ExceptionCheck());
  }
```

（6）添加 TAG 信息

在文件/frameworks/av/media/libmedia/MediaScannerClient.cpp 中，通过函数 addStringTag()添加 TAG 信息。这个 MediaScannerClient 是在 opencore 中的文件 MediaScanner.cpp 中实现的，而文件 android_media_MediaScanner.cpp 中的 MyMediaScannerClient 是从 MediaScannerClient 派生的。函数 addStringTag()的具体实现代码如下所示。

```
status_t MediaScannerClient::addStringTag(const char* name, const char* value)
{
    if (mLocaleEncoding != kEncodingNone) {

        //缓存中不能是 ASCII 字符串
        //如果有则呼叫 handlestringtag 使用 UTF8 直接代替
        bool nonAscii = false;
        const char* chp = value;
        char ch;
        while ((ch = *chp++)) {
            if (ch & 0x80) {
                nonAscii = true;
                break;
            }
        }
//判断 name 和 value 的编码是不是 ASCII 码，不是则保存到 mNames 和 mValues 中
            mNames->push_back(name);
            mValues->push_back(value);
            return OK;
        }
        //其他的失败情形
    }
    //如果字符编码是 ASCII 码，则调用函数 handleStringTag()
    return handleStringTag(name, value);
}
```

（7）JNI 中的环境变量

在 Android 的所有模块的 JNI 层的代码中，会看到很多函数中都有 JNIEnv*类型的参数，例如，文

件/frameworks/base/core/jni/android_hardware_Camera.c 中的如下代码。

```
static void android_hardware_Camera_startPreview(JNIEnv *env, jobject thiz)
{
    LOGV("startPreview");
    sp<Camera> camera = get_native_camera(env, thiz, NULL);
    if (camera == 0) return;
    if (camera->startPreview() != NO_ERROR) {
        jniThrowRuntimeException(env, "startPreview failed");
        return;
    }
}
```

在上述函数中，第一个参数为 JNIEnv *Env，此处的 JNIEnv *类型是一个指向 JNI 环境的指针，JNIEnv *类型在文件 Jni.h 中定义，在此结构体中包含了一些 JNI 中常用到的函数和一组函数指针，C/C++正是通过这些函数指针来操作 Java 函数的。JNIEnv 结构体在文件 jni.h 中定义，具体实现代码如下所示。

```
struct _JNIEnv {
    …
    jint GetVersion()
    { return functions->GetVersion(this); }
    …
    jclass FindClass(const char* name)
    { return functions->FindClass(this, name); }
    …
    void CallVoidMethodA(jobject obj, jmethodID methodID, jvalue* args)
    { functions->CallVoidMethodA(this, obj, methodID, args); }
    …
    jmethodID GetStaticMethodID(jclass clazz, const char* name, const char* sig)
    { return functions->GetStaticMethodID(this, clazz, name, sig); }
    …
}
```

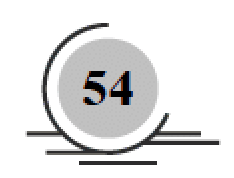
通过上述代码可以发现，正是通过 JNIEnv 指针，才能够调用一些 JNI 环境中的方法。

2.3　分析 Camera 系统的 JNI

本节将以 Camera 系统中的预览功能作为素材，在 Android 源码中详细分析 JNI 机制衔接 Java 层和 C/C++层的方法，剖析 Java 层调用底层代码实现预览功能的具体流程。

2.3.1　Java 层预览接口

本小节将详细介绍 Camera 模块中预览功能的 Java 层的文件路径，以及其中预览函数的功能作用。Camera 中的 Java 层代码在 Camera.java 文件中实现，其详细路径为/Package/apps/camera/src/com/android/



camera/Camera.java。

在文件 Camera.java 中定义了预览相关的函数 startPreview()和 stopPreview()，这是图像预览的入口函数。在文件 Camera.java 中，函数 startPreview()和 stopPreview()的具体实现代码如下所示。

```
        //开始预览
        private void startPreview() {
        if (mPausing || isFinishing()) return;
        mFocusManager.resetTouchFocus();
        mCameraDevice.setErrorCallback(mErrorCallback);
        // If we're previewing already, stop the preview first (this will blank the screen)
        if (mCameraState != PREVIEW_STOPPED) stopPreview();
        setPreviewDisplay(mSurfaceHolder);
        setDisplayOrientation();
        if (!mSnapshotOnIdle) {
            if  (Parameters.FOCUS_MODE_CONTINUOUS_PICTURE.equals(mFocusManager.getFocusMode()))
{
                mCameraDevice.cancelAutoFocus();
            }
            mFocusManager.setAeAwbLock(false);
        }
        //设置 Camera 的参数
        setCameraParameters(UPDATE_PARAM_ALL);
        if (mCameraPreviewThread != null) {
            synchronized (mCameraPreviewThread) {
                mCameraPreviewThread.notify();
            }
        }
        try {
            Log.v(TAG, "startPreview");
            //调用框架层的 Camera 类来实现预览功能
            mCameraDevice.startPreview();
        } catch (Throwable ex) {
            closeCamera();
            throw new RuntimeException("startPreview failed", ex);
        }
        mZoomState = ZOOM_STOPPED;
        setCameraState(IDLE);
        mFocusManager.onPreviewStarted();
        if (mSnapshotOnIdle) {
            mHandler.post(mDoSnapRunnable);
        }
}
//停止预览
private void stopPreview() {
        //判断 Camera 的状态
        if (mCameraDevice != null && mCameraState != PREVIEW_STOPPED) {
            Log.v(TAG, "stopPreview");
            mCameraDevice.cancelAutoFocus();
            mCameraDevice.stopPreview();
```

```
            mFaceDetectionStarted = false;
        }
        //设置 Camera 的状态
        setCameraState(PREVIEW_STOPPED);
        mFocusManager.onPreviewStopped();
}
```

上述代码演示了 Java 层的函数功能，在 Android 的框架层封装了 Camera 的框架层类 Camera，此类的具体路径为 frameworks/base/code/java/android/hardware/Camera.java。

在类 Camera 中声明了很多 native 的方法，例如 startPreview()、stopPreview()，具体声明代码如下。

```
public native final void startPreview();
public native final void stopPreview();
```

上述声明的函数 native 会直接注册到 JNI 中，然后调用 C/C++层的 startPreview()和 stopPreview()函数。

在文件 android_hardware_Camera.cpp 中实现注册 Camera 预览函数的功能，此文件的具体文件路径为 frameworks/base/core/jni/android_hardware_Camera.cpp。

2.3.2 注册预览的 JNI 函数

本小节详细介绍将 Camera 模块的预览功能注册到 JNI 系统的方法。在文件 android_hardware_Camera.cpp 中会将 Camera 模块中的所有接口函数注册到 JNI 系统中，文件 android_hardware_Camera.cpp 中的具体注册代码如下。

```
//初始化 JNI 中的 Java 对象并注册 Camera 模块的 JNI 函数
int register_android_hardware_Camera(JNIEnv *env)
{
    field fields_to_find[] = {
        { "android/hardware/Camera", "mNativeContext", "I", &fields.context },
        { "android/view/Surface", ANDROID_VIEW_SURFACE_JNI_ID,
          "I",&fields.surface },
        { "android/graphics/SurfaceTexture",
          ANDROID_GRAPHICS_SURFACETEXTURE_JNI_ID, "I", &fields.surfaceTexture },
        { "android/hardware/Camera$CameraInfo", "facing", "I",
          &fields.facing },
        { "android/hardware/Camera$CameraInfo", "orientation", "I",
          &fields.orientation },
        { "android/hardware/Camera$Face", "rect", "Landroid/graphics/Rect;",
          &fields.face_rect },
        { "android/hardware/Camera$Face", "score", "I", &fields.face_score },
        { "android/graphics/Rect", "left", "I", &fields.rect_left },
        { "android/graphics/Rect", "top", "I", &fields.rect_top },
        { "android/graphics/Rect", "right", "I", &fields.rect_right },
        { "android/graphics/Rect", "bottom", "I", &fields.rect_bottom },
    };
    if (find_fields(env, fields_to_find, NELEM(fields_to_find)) < 0)
        return -1;
    jclass clazz = env->FindClass("android/hardware/Camera");
```

```
    fields.post_event = env->GetStaticMethodID(clazz, "postEventFromNative",
      "(Ljava/lang/Object;IIILjava/lang/Object;)V");
    if (fields.post_event == NULL) {
        LOGE("Can't find android/hardware/Camera.postEventFromNative");
        return -1;
    }
    clazz = env->FindClass("android/graphics/Rect");
    fields.rect_constructor = env->GetMethodID(clazz, "<init>", "()V");
    if (fields.rect_constructor == NULL) {
        LOGE("Can't find android/graphics/Rect.Rect()");
        return -1;
    }
    clazz = env->FindClass("android/hardware/Camera$Face");
    fields.face_constructor = env->GetMethodID(clazz, "<init>", "()V");
    if (fields.face_constructor == NULL) {
        LOGE("Can't find android/hardware/Camera$Face.Face()");
        return -1;
    }
    // Register native functions
//注册接口函数到 JNI 中
return AndroidRuntime::registerNativeMethods(env, "android/hardware/Camera",
                                        camMethods, NELEM(camMethods));
```

在上述代码中，函数 register_android_hardware_Camera()的功能是初始化 Java 的 Camera 相关的类，并且将接口函数注册到 JNI 中，在文件 android_hardware_Camera.cpp 中，Camera 的函数映射表如下所示。

```
static JNINativeMethod camMethods[] = {
  { "getNumberOfCameras",
    "()I",
    (void *)android_hardware_Camera_getNumberOfCameras },
  { "getCameraInfo",
    "(ILandroid/hardware/Camera$CameraInfo;)V",
    (void*)android_hardware_Camera_getCameraInfo },
  { "native_setup",
    "(Ljava/lang/Object;I)V",
    (void*)android_hardware_Camera_native_setup },
  { "native_release",
    "()V",
    (void*)android_hardware_Camera_release },
  { "setPreviewDisplay",
    "(Landroid/view/Surface;)V",
    (void *)android_hardware_Camera_setPreviewDisplay },
  { "setPreviewTexture",
    "(Landroid/graphics/SurfaceTexture;)V",
    (void *)android_hardware_Camera_setPreviewTexture },
  //开始预览
  { "startPreview",
    "()V",
    (void *)android_hardware_Camera_startPreview },
  //停止预览
```

```
{ "_stopPreview",
  "()V",
  (void *)android_hardware_Camera_stopPreview },
{ "previewEnabled",
  "()Z",
  (void *)android_hardware_Camera_previewEnabled },
{ "setHasPreviewCallback",
  "(ZZ)V",
  (void *)android_hardware_Camera_setHasPreviewCallback },
{ "_addCallbackBuffer",
  "([BI)V",
  (void *)android_hardware_Camera_addCallbackBuffer },
{ "native_autoFocus",
  "()V",
  (void *)android_hardware_Camera_autoFocus },
{ "native_cancelAutoFocus",
  "()V",
  (void *)android_hardware_Camera_cancelAutoFocus },
{ "native_takePicture",
  "(I)V",
  (void *)android_hardware_Camera_takePicture },
{ "native_setParameters",
  "(Ljava/lang/String;)V",
  (void *)android_hardware_Camera_setParameters },
{ "native_getParameters",
  "()Ljava/lang/String;",
  (void *)android_hardware_Camera_getParameters },
{ "reconnect",
  "()V",
  (void*)android_hardware_Camera_reconnect },
{ "lock",
  "()V",
  (void*)android_hardware_Camera_lock },
{ "unlock",
  "()V",
  (void*)android_hardware_Camera_unlock },
{ "startSmoothZoom",
  "(I)V",
  (void *)android_hardware_Camera_startSmoothZoom },
{ "stopSmoothZoom",
  "()V",
  (void *)android_hardware_Camera_stopSmoothZoom },
{ "setDisplayOrientation",
  "(I)V",
  (void *)android_hardware_Camera_setDisplayOrientation },
{ "_startFaceDetection",
  "(I)V",
  (void *)android_hardware_Camera_startFaceDetection },
{ "_stopFaceDetection",
```

```
    "()V",
    (void *)android_hardware_Camera_stopFaceDetection},
};
```

根据上述代码可以看到，有了这个函数映射表，则 Camera 的 Java 层接口可以调用到 C/C++层的接口函数，C/C++层的预览函数指针名为 android_hardware_Camera_startPreview()、android_hardware_Camera_stopPreview()，这两个函数会进而调用到 C/C++层的函数，在文件 android_hardware_Camera.cpp 中，其具体代码如下所示。

```
static void android_hardware_Camera_startPreview(JNIEnv *env, jobject thiz)
{
    ALOGV("startPreview");
    //获得 C/C++层的 Camera 指针
    sp<Camera> camera = get_native_camera(env, thiz, NULL);
    if (camera == 0) return;
    //调用 C/C++层的 startPreview()
    if (camera->startPreview() != NO_ERROR) {
        jniThrowRuntimeException(env, "startPreview failed");
        return;
    }
}
static void android_hardware_Camera_stopPreview(JNIEnv *env, jobject thiz)
{
    ALOGV("stopPreview");
    //获得 C/C++层的 Camera 指针
    sp<Camera> c = get_native_camera(env, thiz, NULL);
    if (c == 0) return;
    //调用 C/C++层的 stopPreview()
    c->stopPreview();
}
```

2.3.3　C/C++层的预览函数

Camera 模块的 C/C++层文件路径为/frameworks/av/camera/Camera.cpp，具体的实现代码如下所示。

```
//开始预览
status_t Camera::startPreview()
{
    ALOGV("startPreview");
    sp <ICamera> c = mCamera;
    if (c == 0) return NO_INIT;
    //调用其他 so 库的 startPreview()
    return c->startPreview();
}
//停止预览
void Camera::stopPreview()
{
    ALOGV("stopPreview");
    sp <ICamera> c = mCamera;
```

```
    if (c == 0) return;
    //调用其他 so 库的 stopPreview()
    c->stopPreview();
}
```

通过上述代码发现，文件 Camera.cpp 的功能是实现 Camera 模块中的 C/C++层，然后继续调用更加底层的预览的实现代码。

2.4 Java 与 JNI 基本数据类型转换

在 Android 5.0 中，Java 与 JNI 基本数据类型转换信息如表 2-2 所示。

表 2-2 基本数据类型的转换关系

Java	Native 类型	本地 C 类型	字 长
boolean	jboolean	无符号	8 位
byte	jbyte	无符号	8 位
char	jchar	无符号	16 位
short	jshort	有符号	16 位
int	jint	有符号	32 位
long	jlong	有符号	64 位
float	jfloat	有符号	32 位
double	jdouble	有符号	64 位

数组类型的对应关系如表 2-3 所示。

表 2-3 数组数据类型的对应关系

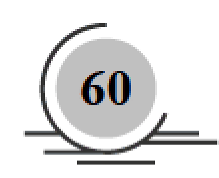
字 符	Java 类型	C 类型
[I	jintArray	int[]
[F	jfloatArray	float[]
[B	jbyteArray	byte[]
[C	jshortArray	short[]
[D	jdoubleArray	double[]
[J	jlongArray	long[]
[S	jshortArray	short[]
[Z	jbooleanArray	boolean[]

对象数据类型的对应关系如表 2-4 所示。

表 2-4 对象数据类型的对应关系

对 象	Java 类型	C 类型
Ljava/lang/String	String	jstring
Ljava/net/Socket	Socket	jobject

引用数据类型的转换关系如图 2-5 所示。



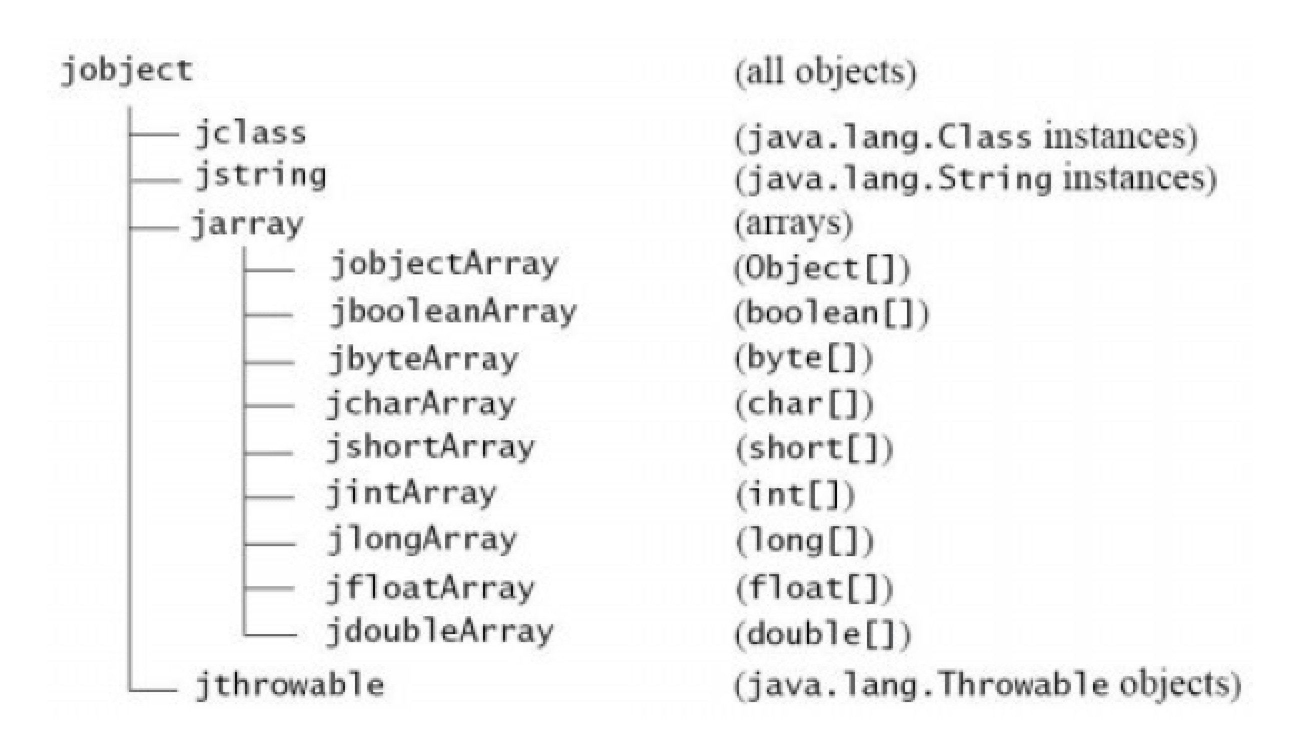


图 2-5　引用数据类型的转换关系

2.5　JNIEnv 接口

在 Android 5.0 中，Native Method（本地方法）中的 JNIEnv 作为第一个参数被传入。JNIEnv 的内部结构如图 2-6 所示。

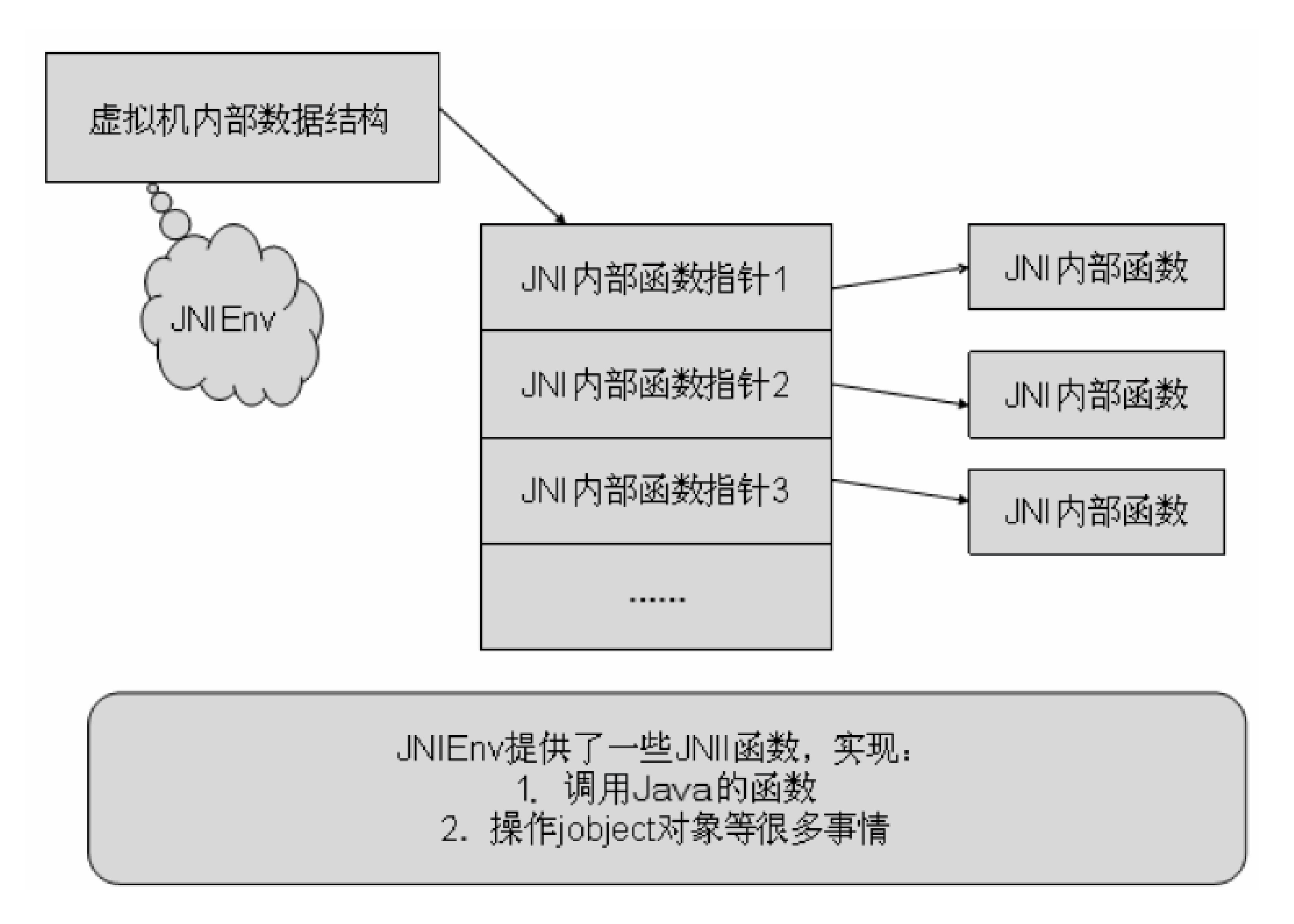


图 2-6　JNIEnv 的内部结构

当 JNIEnv 不作为参数传入时，JNI 提供了如下两个函数来获得 JNIEnv 接口。

☑　(*jvm)->AttachCurrentThread(jvm, (void**)&env, NULL)

☑　(*jvm)->GetEnv(jvm, (void**)&env, JNI_VERSION_1_2)

上述两个函数都利用 Java VM 接口获得了 JNIEnv 接口，并且 JNI 可以将获得的 JNIEnv 封装成一个函数。

```
JNIEnv* JNU_GetEnv()
{
    JNIEnv* env;
```

```
    (*g_jvm)->GetEnv(g_jvm, (void**)&env, JNI_VERSION_1_2);
    return env;
}
```

Java 通过 JNI 机制调用 C/C++写的 Native 程序，C/C++开发的 Native 程序需要遵循一定的 JNI 规范。例如，下面就是一个 JNI 函数声明的例子。

```
JNIEXPORT jint JNICALL Java_jnitest_MyTest_test
  (JNIEnv * env, jobject obj, jint arg0);
```

JVM 负责从 Java Stack 转入 C/C++ Native Stack。当 Java 进入 JNI 调用，除了函数本身的参数（arg0）外会多出两个参数：JNIEnv 指针和 jobject 指针。其中，JNIEnv 指针是 JVM 创建的，被 Native 的 C/C++ 方法用来操纵 Java 执行栈中的数据，例如 Java Class、Java Method 等。

首先，JNI 对于 JNIEnv 的使用提供了两种语法，分别是 C 语法以及 C++语法。其中，C 语法是：

```
jsize len = (*env)->GetArrayLength(env,array);
```

C++语法是：

```
jsize len =env->GetArrayLength(array);
```

因为 C 语言并不支持对象的概念，所以 C 语法中需要把 env 作为第一个参数传入，这类似于 C++ 的隐式参数 this 指针。

另外，在使用 JNIEnv 接口时，需要遵循如下两个设计原则。

（1）JNIEnv 指针被设计成了 Thread Local Storage（TLS）变量，也就是说每一个 Thread、JNIEnv 变量都有独立的 Copy，不能把 Thead#1 使用的 JNIEnv 传给 Thread#2 使用。

（2）在 JNIEnv 中定义了一组函数指针，C/C++ Native 程序是通过这些函数指针操纵 Java 数据。这样设计的好处是，C/C++ 程序不需要依赖任何函数库或者 DLL。由于 JVM 可能由不同的厂商实现，不同厂商有自己不同的 JNI 实现，如果要求这些厂商暴露约定好的一些头文件和库，这不是灵活的设计。而且使用函数指针表的另外一个好处是，JVM 可以根据启动参数动态替换 JNI 实现。

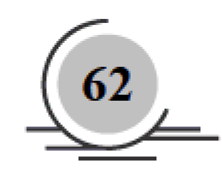
在 jint JNI_OnLoad(JavaVM* vm, void* reserved)的整个进程只有一个 JavaVM 对象，可以保存并在任何地方使用。利用 JavaVM 中的 AttachCurrentThread()函数，即可得到这个线程的 JNIEnv 结构体，利用 DetachCurrentThread()释放相应资源。

2.6　开发 JNI 程序

在现实开发应用的过程中，可以对 Android 操作系统进行适当的修改以增加各种自定义功能，这样可以满足用户的特定需求。在 Android 系统中，Dalvik VM 中的应用程序使用 JNI（Java Native Interface）来调用 C/C++开发的共享库。本节将详细讲解开发一个自己的 JNI 程序的方法。

2.6.1　开发 JNI 程序的步骤

在 Android 5.0 中，开发 JNI 程序的一般操作步骤如下所示。



（1）编写 Java 中的调用类。
（2）用 javah 生成 C/C++原生函数的头文件。
（3）在 C/C++中调用需要的其他函数功能实现原生函数，原则上可以调用任何资源。
（4）将项目依赖的所有原生库和资源加入到 Java 项目的 java.library.path。
（5）生成 Java 程序。

2.6.2　开发一个自己的 JNI 程序

（1）打开 Eclipse，新建一个名为 testJni 的工程名。
（2）在 Activity 中添加如下代码。

```
package com.aaa.jni;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.TextView;
import android.support.v4.app.NavUtils;
public class TestJni extends Activity {
	@Override
	public void onCreate(Bundle savedInstanceState) {
		super.onCreate(savedInstanceState);
	}
      //natvie 必须声明，用于生成 C/C++代码
	public native String hello();

	static{
		  System.loadLibrary("testJni");
	}

}
```

编译后的文件被保存在 bin 目录下，通过 javah 命令生成 C/C++的头文件。此时会在项目目录下生成文件 jni/com_aaa_jni_TestJni.h，头文件的具体代码如下所示。

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class com_aaa_jni_TestJni */

#ifndef _Included_com_aaa_jni_TestJni
#define _Included_com_aaa_jni_TestJni
#ifdef __cplusplus
extern "C" {
#endif
JNIEXPORT jstring JNICALL Java_com_aaa_jni_TestJni_hello
  (JNIEnv *, jobject);
```

```
#ifdef __cplusplus
}
#endif
#endif
```

（3）根据头文件编写 C 代码，具体代码如下所示。

```
#include <string.h>
#include <jni.h>
jstring
Java_com_aaa_jni_TestJni_hello
  (JNIEnv* env, jobject thiz){
		return (*env)->NewStringUTF(env, "哈哈完成自动化编译 !");
}
```

（4）编写文件 Android.mk，可以直接从 NDK 的 samples 下 hello-jni 的 JNI 文件下复制该文件，具体代码如下所示。

```
# Copyright (C) 2009 The Android Open Source Project
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License
#
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := testJni
LOCAL_SRC_FILES := testJni.c

include $(BUILD_SHARED_LIBRARY)
```

在此只需要修改 LOCAL_MODULE 和 LOCAL_SRC_FILES 即可。

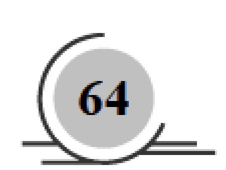
☑ LOCAL_MODULE：模块描述信息，用于给 Java 调用的模块名，会生成对应的 libtestJni.so 文件。

☑ LOCAL_SRC_FILES：源文件，多个文件之间可以用空格隔开。

（5）开始编译生成.so 文件。打开 gnustep 工具的命令窗口，在项目下输入如下命令即可自动生成 libs/armeabi/libtestJni.so 文件，如图 2-7 所示。

```
$NDK/ndk-build
```

（6）编写 Java 调用代码，具体代码如下所示。



```
package com.aaa.jni;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.TextView;
import android.support.v4.app.NavUtils;
public class TestJni extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView tv = new TextView(this);
        tv.setText(hello()); //这里的 hello()就是调用代码
        setContentView(tv);
    }

    public native String hello();

    static{
            System.loadLibrary("testJni");
    }

}
```

这样便成功实现了自己编写 JNI 程序并调用的目标，编译运行后的效果如图 2-8 所示。

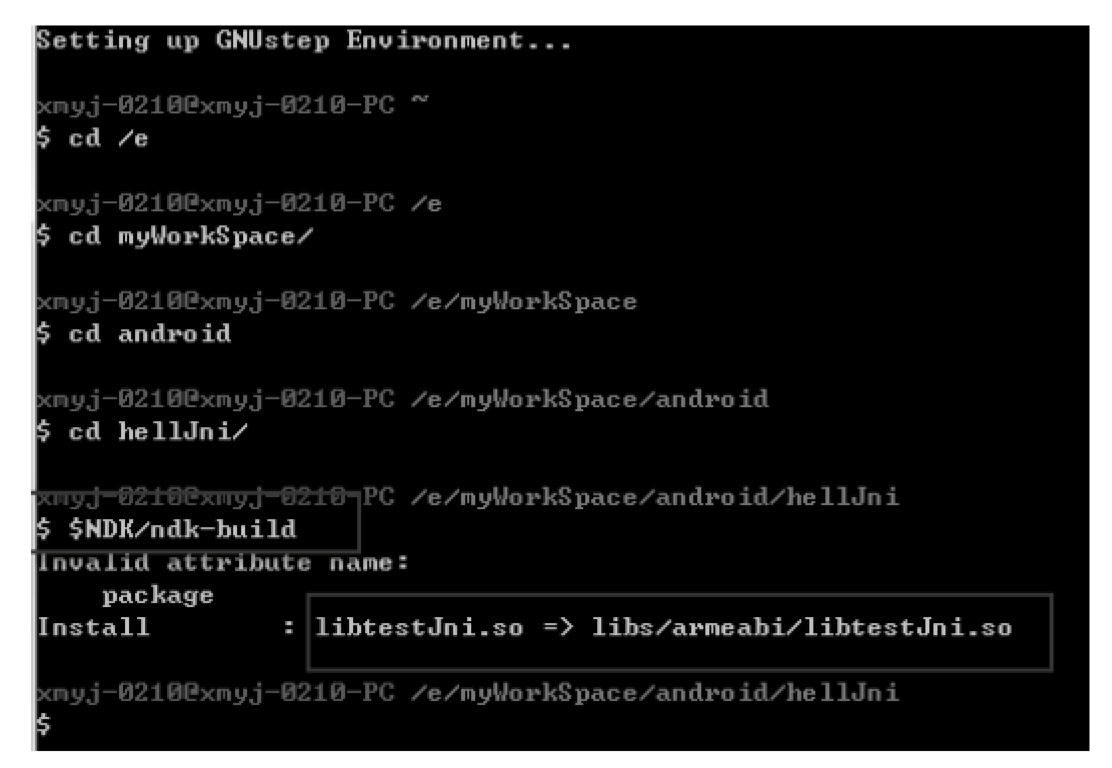


图 2-7　开始编译生成.so 文件



图 2-8　执行效果

第 3 章　内存系统架构详解

内存（Memory）也被称为内存储器，其作用是暂时存放 CPU 中的运算数据，以及与硬盘等外部存储器交换的数据。内存是计算机中重要的部件之一，是用户任务与 CPU 处理进行沟通的桥梁。CPU 会把计算机运行中需要运算的数据放到内存中进行运算处理，并将运算完成后的结果传送出来。本章将详细分析 Android 5.0 内存系统的基本源码。

3.1　分析 Android 的进程通信机制

要想实现对 Android 系统内存的优化，需要首先了解 Android 的内存系统，了解内存控制进程运行的机制。本节将探讨分析 Android 的进程通信机制。

3.1.1　IPC 机制介绍

在 Android 5.0 系统中，每一个应用程序都是由一些 Activity 和 Service 组成的，一般 Service 运行在独立的进程中，而 Activity 可能运行在同一个进程中，也有可能运行在不同的进程中。那么不在同一个进程的 Activity 或者 Service 之间究竟是如何通信的呢？Android 系统通过 Binder 进程间通信机制来实现这个功能。

其实 Binder 并不是 Android 提出来的一套新的进程间通信机制，它是基于 OpenBinder 来实现的。Binder 是一种进程间通信机制，Android 系统的 Binder 机制由如下系统组件组成。

- ☑ Client
- ☑ Server
- ☑ Service Manager
- ☑ Binder 驱动程序

其中，Client、Server 和 Service Manager 在用户空间运行，Binder 驱动程序在内核空间中运行。Binder 就是一种把这 4 个组件“黏合”在一起的“黏结剂”。其中的核心组件便是 Binder 驱动程序，Service Manager 提供了辅助管理的功能，Client 和 Server 正是在 Binder 驱动和 Service Manager 提供的基础设施上实现 Client/Server 之间的通信。Service Manager 和 Binder 驱动已经在 Android 平台中实现完毕，开发者只要按照规范实现自己的 Client 和 Server 组件即可。对于初学者来说，Android 系统的 Binder 机制是最难理解的，而 Binder 机制无论从系统开发还是应用开发的角度来看，都是 Android 系统中最重要的组成，所以很有必要深入了解 Binder 的工作方式。要深入了解 Binder 的工作方式，最好的方式就是阅读 Binder 相关的源代码。

要想深入理解 Binder 机制，必须了解 Binder 在用户空间的 3 个组件 Client、Server 和 Service Manager 之间的相互关系，并了解内核空间中 Binder 驱动程序的数据结构和设计原理。具体来说，Android 系统 Binder 机制中的 4 个组件 Client、Server、Service Manager 和 Binder 驱动程序的关系如图 3-1 所示。

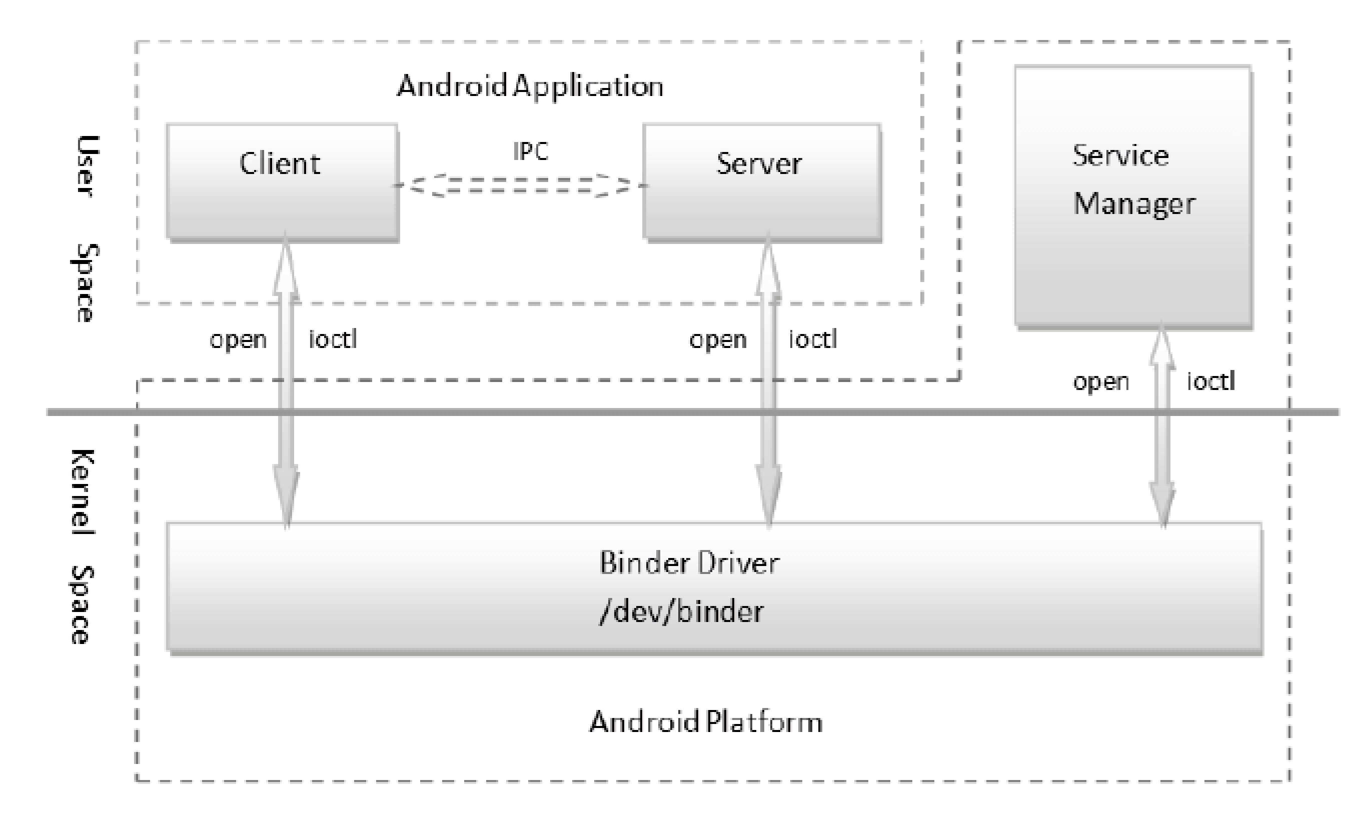


图 3-1　组件 Client、Server、Service Manager 和 Binder 驱动程序的关系

图 3-1 中构成组件的具体说明如下所示。

（1）Client、Server 和 Service Manager：在用户空间中实现，Binder 驱动程序在内核空间中实现。

（2）Binder 驱动程序和 Service Manager：已经在 Android 平台中实现，开发者只需要在用户空间实现自己的 Client 和 Server 即可。

（3）Binder 驱动程序提供的设备文件/dev/binder：负责与用户空间进行交互，Client、Server 和 Service Manager 通过文件操作函数 open()和 ioctl()与 Binder 驱动程序进行通信。

（4）Service Manager：是一个用来管理 Server 的保护进程，并向 Client 提供查询 Server 接口的能力。

3.1.2　Service Manager 是 Binder 机制的上下文管理者

分析 Android 5.0 的 Binder 源代码，Service Manager 是整个 Binder 机制的保护进程，用来管理开发者创建的各种 Server，并且向 Client 提供查询 Server 远程接口的功能。因为 Service Manager 组件的功能是用来管理 Server 并且向 Client 提供查询 Server 远程接口，所以 Service Manager 必然要和 Server 以及 Client 进行通信。Service Manager、Client 和 Server 三者分别是运行在独立的进程当中的，这样它们之间的通信也属于进程间的通信，而且也是采用 Binder 机制进行进程间通信。因此，Service Manager 在充当 Binder 机制的保护进程角色的同时，也在充当 Server 的角色，是一种特殊的 Server。

在 Android 5.0 中，Service Manager 在用户空间的源代码位于 frameworks/base/cmds/servicemanager 目录下，主要是由文件 binder.h、binder.c 和 service_manager.c 组成。Service Manager 在 Binder 机制中的基本执行流程如图 3-2 所示。

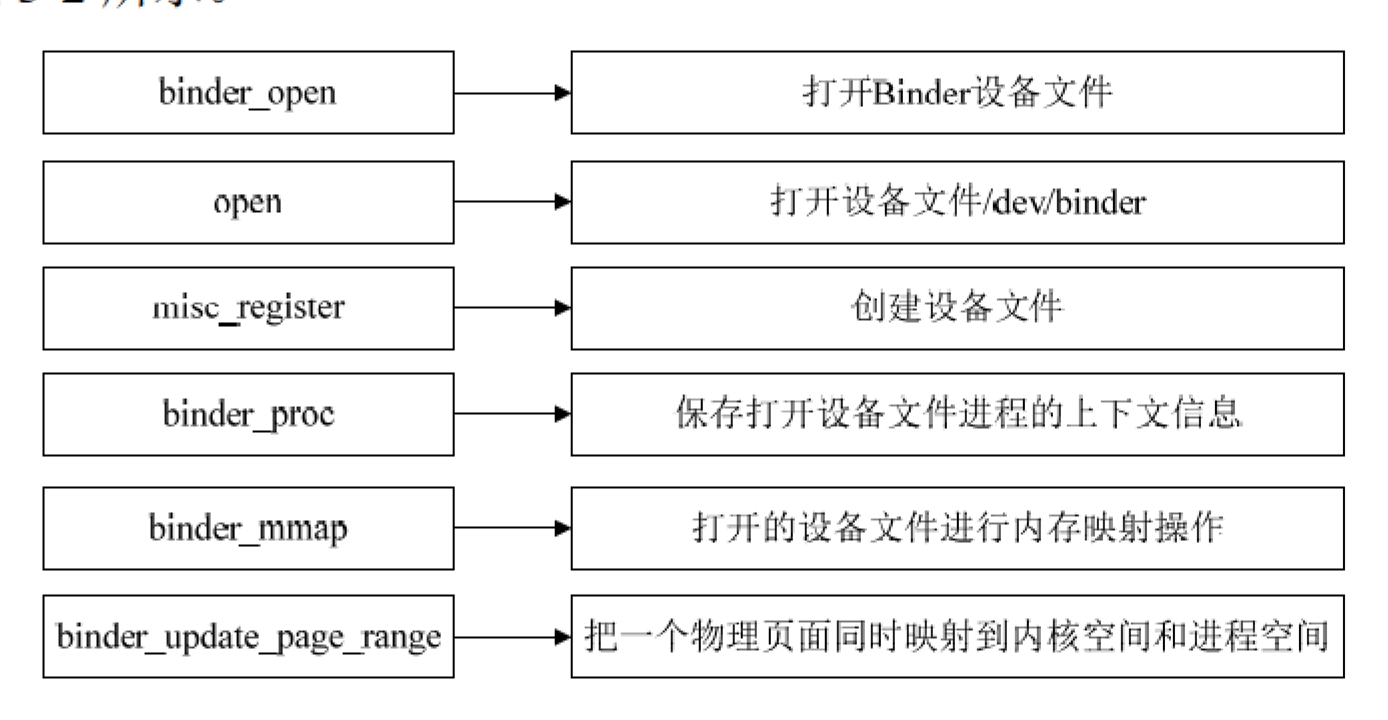


图 3-2　Service Manager 在 Binder 机制中的基本执行流程

在 Android 5.0 中，Service Manager 的入口函数 main()位于文件 service_manager.c 中，具体代码如下所示。

```
int main(int argc, char **argv){
    struct binder_state *bs;
    void *svcmgr = BINDER_SERVICE_MANAGER;
    bs = binder_open(128*1024);
    if (binder_become_context_manager(bs)) {
        LOGE("cannot become context manager (%s)\n", strerror(errno));
        return -1;
    }
    svcmgr_handle = svcmgr;
    binder_loop(bs, svcmgr_handler);
    return 0;
}
```

在上述代码中，函数 main()主要有如下 3 个功能。

- ☑ 打开 Binder 设备文件。
- ☑ 告诉 Binder 驱动程序自己是 Binder 上下文管理者，即前面提及的保护进程。
- ☑ 进入一个无穷循环，充当 Server 的角色，等待 Client 的请求。

在分析上述 3 个功能之前，先来看一下这里用到的结构体 binder_state、宏 BINDER_SERVICE_MANAGER 的定义。结构体 binder_state 在文件 frameworks/base/cmds/servicemanager/binder.c 中定义，具体代码如下所示。

```
struct binder_state {
    int fd;
    void *mapped;
    unsigned mapsize;
};
```

其中，fd 表示文件描述符，即表示打开的/dev/binder 设备文件描述符；mapped 表示把设备文件/dev/binder 映射到进程空间的起始地址；mapsize 表示上述内存映射空间的大小。

宏 BINDER_SERVICE_MANAGER 在文件 frameworks/base/cmds/servicemanager/binder.h 中定义，代码如下。

```
/* the one magic object */
#define BINDER_SERVICE_MANAGER ((void*) 0)
```

这表示 Service Manager 的句柄为 0，Binder 通信机制使用句柄来代表远程接口。

函数首先打开 Binder 设备文件的操作函数 binder_open()，此函数的定义位于文件 frameworks/base/cmds/servicemanager/binder.c 中，具体代码如下所示。

```
struct binder_state *binder_open(unsigned mapsize){
    struct binder_state *bs;
    bs = malloc(sizeof(*bs));
    if (!bs) {
        errno = ENOMEM;
        return 0;
```

```
	}
	bs->fd = open("/dev/binder", O_RDWR);
	if (bs->fd < 0) {
		fprintf(stderr,"binder: cannot open device (%s)\n",
				strerror(errno));
		goto fail_open;
	}
	bs->mapsize = mapsize;
	bs->mapped = mmap(NULL, mapsize, PROT_READ, MAP_PRIVATE, bs->fd, 0);
	if (bs->mapped == MAP_FAILED) {
		fprintf(stderr,"binder: cannot map device (%s)\n",
				strerror(errno));
		goto fail_map;
	}
		/* TODO: check version */
	return bs;
fail_map:
	close(bs->fd);
fail_open:
	free(bs);
	return 0;
}
```

通过文件操作函数 open()打开设备文件/dev/binder，此设备文件是在 Binder 驱动程序模块初始化时创建的。接下来先看一下这个设备文件的创建过程，进入到 kernel/common/drivers/staging/android 目录，打开文件 binder.c，可以看到如下模块初始化入口代码 binder_init。

```
static struct file_operations binder_fops = {
	.owner = THIS_MODULE,
	.poll = binder_poll,
	.unlocked_ioctl = binder_ioctl,
	.mmap = binder_mmap,
	.open = binder_open,
	.flush = binder_flush,
	.release = binder_release,
};

static struct miscdevice binder_miscdev = {
	.minor = MISC_DYNAMIC_MINOR,
	.name = "binder",
	.fops = &binder_fops
};

static int __init binder_init(void)
{
	int ret;

	binder_proc_dir_entry_root = proc_mkdir("binder", NULL);
	if (binder_proc_dir_entry_root)
		binder_proc_dir_entry_proc = proc_mkdir("proc", binder_proc_dir_entry_root);
	ret = misc_register(&binder_miscdev);
```

```
    if (binder_proc_dir_entry_root) {
        create_proc_read_entry("state", S_IRUGO, binder_proc_dir_entry_root, binder_read_proc_state, NULL);
        create_proc_read_entry("stats", S_IRUGO, binder_proc_dir_entry_root, binder_read_proc_stats, NULL);
        create_proc_read_entry("transactions", S_IRUGO, binder_proc_dir_entry_root, binder_read_proc_
transactions, NULL);
        create_proc_read_entry("transaction_log", S_IRUGO, binder_proc_dir_entry_root, binder_read_proc_
transaction_log, &binder_transaction_log);
        create_proc_read_entry("failed_transaction_log", S_IRUGO, binder_proc_dir_entry_root, binder_read_
proc_transaction_log, &binder_transaction_log_failed);
    }
    return ret;
}
device_initcall(binder_init);
```

在函数 misc_register()中实现了创建设备文件的功能，并实现了 misc 设备的注册工作，在/proc 目录中创建了各种 Binder 相关的文件供用户访问。通过如下执行语句即可进入到 Binder 驱动程序的 binder_open()函数。

```
bs->fd = open("/dev/binder", O_RDWR);
```

Binder 驱动程序函数 binder_open()的主要作用是创建一个名为 binder_proc 的数据结构，用此数据结构来保存打开设备文件/dev/binder 的进程的上下文信息，并且将这个进程上下文信息保存在打开文件结构 file 的私有数据成员变量 private_data 中。函数 binder_open()的实现代码如下所示。

```
static int binder_open(struct inode *nodp, struct file *filp)
{
    struct binder_proc *proc;

    if (binder_debug_mask & BINDER_DEBUG_OPEN_CLOSE)
        printk(KERN_INFO "binder_open: %d:%d\n", current->group_leader->pid, current->pid);

    proc = kzalloc(sizeof(*proc), GFP_KERNEL);
    if (proc == NULL)
        return -ENOMEM;
    get_task_struct(current);
    proc->tsk = current;
    INIT_LIST_HEAD(&proc->todo);
    init_waitqueue_head(&proc->wait);
    proc->default_priority = task_nice(current);
    mutex_lock(&binder_lock);
    binder_stats.obj_created[BINDER_STAT_PROC]++;
    hlist_add_head(&proc->proc_node, &binder_procs);
    proc->pid = current->group_leader->pid;
    INIT_LIST_HEAD(&proc->delivered_death);
    filp->private_data = proc;
    mutex_unlock(&binder_lock);

    if (binder_proc_dir_entry_proc) {
        char strbuf[11];
        snprintf(strbuf, sizeof(strbuf), "%u", proc->pid);
```

```
        remove_proc_entry(strbuf, binder_proc_dir_entry_proc);
        create_proc_read_entry(strbuf, S_IRUGO, binder_proc_dir_entry_proc, binder_read_proc_proc, proc);
    }
    return 0;
}
```

而结构体 struct binder_proc 也被定义在文件 kernel/common/drivers/staging/android/binder.c 中，此结构体的成员比较多，其中最为重要的是如下 4 个成员变量。

- ☑ threads
- ☑ nodes
- ☑ refs_by_desc
- ☑ refs_by_node

上述 4 个成员变量都是表示红黑树的节点，即 binder_proc 分别挂在 4 个红黑树下，具体说明如下所示。

- ☑ threads 树：用来保存 binder_proc 进程内用于处理用户请求的线程，其最大数量由 max_threads 来决定。
- ☑ nodes 树：用来保存 binder_proc 进程内的 Binder 实体。
- ☑ refs_by_desc 树和 refs_by_node 树：用来保存 binder_proc 进程内的 Binder 引用，即引用的其他进程的 Binder 实体，分别用两种方式来组织红黑树，一种是以句柄作为 key 值来组织，一种是以引用的实体节点的地址值作为 key 值来组织，二者意义相同，只不过是为了内部查找方便而用两个红黑树来表示。

结构体 struct binder_proc 的具体代码如下所示。

```
struct binder_proc {
    struct hlist_node proc_node;
    struct rb_root threads;
    struct rb_root nodes;
    struct rb_root refs_by_desc;
    struct rb_root refs_by_node;
    int pid;
    struct vm_area_struct *vma;
    struct task_struct *tsk;
    struct files_struct *files;
    struct hlist_node deferred_work_node;
    int deferred_work;
    void *buffer;
    ptrdiff_t user_buffer_offset;
    struct list_head buffers;
    struct rb_root free_buffers;
    struct rb_root allocated_buffers;
    size_t free_async_space;
    struct page **pages;
    size_t buffer_size;
    uint32_t buffer_free;
    struct list_head todo;
    wait_queue_head_t wait;
```

```
    struct binder_stats stats;
    struct list_head delivered_death;
    int max_threads;
    int requested_threads;
    int requested_threads_started;
    int ready_threads;
    long default_priority;
};
```

这样，打开设备文件/dev/binder 的操作就完成了，接下来需要对打开的设备文件进行内存映射操作。

```
bs->mapped = mmap(NULL, mapsize, PROT_READ, MAP_PRIVATE, bs->fd, 0);
```

对应 Binder 驱动程序的是函数 binder_mmap()，首先通过 filp->private_data 得到在打开设备文件/dev/binder 时创建的结构 binder_proc。内存映射信息放在 vma 参数中。此处 vma 的数据类型是结构 vm_area_struct，表示的是一块连续的虚拟地址空间区域。另外，结构体 vm_struct 表示一块连续的虚拟地址空间区域。函数 binder_mmap()的具体实现代码如下所示。

```
static int binder_mmap(struct file *filp, struct vm_area_struct *vma)
{
    int ret;
    struct vm_struct *area;
    struct binder_proc *proc = filp->private_data;
    const char *failure_string;
    struct binder_buffer *buffer;
    if ((vma->vm_end - vma->vm_start) > SZ_4M)
        vma->vm_end = vma->vm_start + SZ_4M;
    if (binder_debug_mask & BINDER_DEBUG_OPEN_CLOSE)
        printk(KERN_INFO
            "binder_mmap: %d %lx-%lx (%ld K) vma %lx pagep %lx\n",
            proc->pid, vma->vm_start, vma->vm_end,
            (vma->vm_end - vma->vm_start) / SZ_1K, vma->vm_flags,
            (unsigned long)pgprot_val(vma->vm_page_prot));
    if (vma->vm_flags & FORBIDDEN_MMAP_FLAGS) {
        ret = -EPERM;
        failure_string = "bad vm_flags";
        goto err_bad_arg;
    }
    vma->vm_flags = (vma->vm_flags | VM_DONTCOPY) & ~VM_MAYWRITE;

    if (proc->buffer) {
        ret = -EBUSY;
        failure_string = "already mapped";
        goto err_already_mapped;
    }

    area = get_vm_area(vma->vm_end - vma->vm_start, VM_IOREMAP);
    if (area == NULL) {
        ret = -ENOMEM;
        failure_string = "get_vm_area";
        goto err_get_vm_area_failed;
```

```
	}
	proc->buffer = area->addr;
	proc->user_buffer_offset = vma->vm_start - (uintptr_t)proc->buffer;

#ifdef CONFIG_CPU_CACHE_VIPT
	if (cache_is_vipt_aliasing()) {
		while (CACHE_COLOUR((vma->vm_start ^ (uint32_t)proc->buffer))) {
			printk(KERN_INFO "binder_mmap: %d %lx-%lx maps %p bad alignment\n", proc->pid, vma->
vm_start, vma->vm_end, proc->buffer);
			vma->vm_start += PAGE_SIZE;
		}
	}
#endif
	proc->pages = kzalloc(sizeof(proc->pages[0]) * ((vma->vm_end - vma->vm_start) PAGE_SIZE), GFP_
KERNEL);
	if (proc->pages == NULL) {
		ret = -ENOMEM;
		failure_string = "alloc page array";
		goto err_alloc_pages_failed;
	}
	proc->buffer_size = vma->vm_end - vma->vm_start;

	vma->vm_ops = &binder_vm_ops;
	vma->vm_private_data = proc;

	if (binder_update_page_range(proc, 1, proc->buffer, proc->buffer + PAGE_SIZE, vma)) {
		ret = -ENOMEM;
		failure_string = "alloc small buf";
		goto err_alloc_small_buf_failed;
	}
	buffer = proc->buffer;
	INIT_LIST_HEAD(&proc->buffers);
	list_add(&buffer->entry, &proc->buffers);
	buffer->free = 1;
	binder_insert_free_buffer(proc, buffer);
	proc->free_async_space = proc->buffer_size / 2;
	barrier();
	proc->files = get_files_struct(current);
	proc->vma = vma;

	/*printk(KERN_INFO "binder_mmap: %d %lx-%lx maps %p\n", proc->pid, vma->vm_start, vma->vm_end,
proc->buffer);*/
	return 0;

err_alloc_small_buf_failed:
	kfree(proc->pages);
	proc->pages = NULL;
err_alloc_pages_failed:
	vfree(proc->buffer);
	proc->buffer = NULL;
err_get_vm_area_failed:
```

```
err_already_mapped:
err_bad_arg:
	printk(KERN_ERR "binder_mmap: %d %lx-%lx %s failed %d\n", proc->pid, vma->vm_start, vma->vm_end,
failure_string, ret);
	return ret;
}
```

接下来分析 binder_proc 结构体中的如下成员变量。

☑ buffer：是一个 void*指针，表示要映射的物理内存在内核空间中的起始位置。

☑ buffer_size：是一个 size_t 类型的变量，表示要映射的内存的大小。

☑ pages：是一个 struct page*类型的数组，struct page 是用来描述物理页面的数据结构。

☑ user_buffer_offset：是一个 ptrdiff_t 类型的变量，表示的是内核使用的虚拟地址与进程使用的虚拟地址之间的差值，即如果某个物理页面在内核空间中对应的虚拟地址是 addr，那么这个物理页面在进程空间对应的虚拟地址就为如下格式。

```
addr + user_buffer_offset
```

接下来还需要看一下 Binder 驱动程序管理内存映射地址空间的方法，即如何管理 buffer ~ (buffer + buffer_size)这段地址空间，这个地址空间被划分为一段一段来管理，每一段是用结构体 binder_buffer 来描述的，具体代码如下所示。

```
struct binder_buffer {
	struct list_head entry; /* free and allocated entries by addesss */
	struct rb_node rb_node; /* free entry by size or allocated entry */
				/* by address */
	unsigned free : 1;
	unsigned allow_user_free : 1;
	unsigned async_transaction : 1;
	unsigned debug_id : 29;
	struct binder_transaction *transaction;
	struct binder_node *target_node;
	size_t data_size;
	size_t offsets_size;
	uint8_t data[0];
};
```

每一个 binder_buffer 通过其成员 entry 按从低地址到高地址连入到 struct binder_proc 中的 buffers 表示的链表中，并且每一个 binder_buffer 又分为正在使用的和空闲的，通过 free 成员变量来区分，空闲的 binder_buffer 借助变量 rb_node 进入 struct binder_proc 中的 free_buffers 表示的红黑树中。而那些正在使用的 binder_buffer 则通过成员变量 rb_node 连入到 binder_proc 中的 allocated_buffers 表示的红黑树中去。这样做是为了方便查询和维护这块地址空间。

继续分析函数 binder_update_page_range()，了解 Binder 驱动程序是如何实现把一个物理页面同时映射到内核空间和进程空间的。具体实现代码如下所示。

```
static int binder_update_page_range(struct binder_proc *proc, int allocate,
	void *start, void *end, struct vm_area_struct *vma)
{
	void *page_addr;
```

```
unsigned long user_page_addr;
struct vm_struct tmp_area;
struct page **page;
struct mm_struct *mm;
if (binder_debug_mask & BINDER_DEBUG_BUFFER_ALLOC)
	printk(KERN_INFO "binder: %d: %s pages %p-%p\n",
		proc->pid, allocate ? "allocate" : "free", start, end);
if (end <= start)
	return 0;
if (vma)
	mm = NULL;
else
	mm = get_task_mm(proc->tsk);
if (mm) {
	down_write(&mm->mmap_sem);
	vma = proc->vma;
}
if (allocate == 0)
	goto free_range;
if (vma == NULL) {
	printk(KERN_ERR "binder: %d: binder_alloc_buf failed to "
		"map pages in userspace, no vma\n", proc->pid);
	goto err_no_vma;
}
for (page_addr = start; page_addr < end; page_addr += PAGE_SIZE) {
	int ret;
	struct page **page_array_ptr;
	page = &proc->pages[(page_addr - proc->buffer) / PAGE_SIZE];
	BUG_ON(*page);
	*page = alloc_page(GFP_KERNEL | __GFP_ZERO);
	if (*page == NULL) {
		printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
			"for page at %p\n", proc->pid, page_addr);
		goto err_alloc_page_failed;
	}
	tmp_area.addr = page_addr;
	tmp_area.size = PAGE_SIZE + PAGE_SIZE /* guard page? */;
	page_array_ptr = page;
	ret = map_vm_area(&tmp_area, PAGE_KERNEL, &page_array_ptr);
	if (ret) {
		printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
			"to map page at %p in kernel\n",
			proc->pid, page_addr);
		goto err_map_kernel_failed;
	}
	user_page_addr =
		(uintptr_t)page_addr + proc->user_buffer_offset;
	ret = vm_insert_page(vma, user_page_addr, page[0]);
	if (ret) {
		printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
			"to map page at %lx in userspace\n",
```

```
                    proc->pid, user_page_addr);
                goto err_vm_insert_page_failed;
            }
            /* vm_insert_page does not seem to increment the refcount */
        }
        if (mm) {
            up_write(&mm->mmap_sem);
            mmput(mm);
        }
        return 0;
free_range:
        for (page_addr = end - PAGE_SIZE; page_addr >= start;
             page_addr -= PAGE_SIZE) {
            page = &proc->pages[(page_addr - proc->buffer) / PAGE_SIZE];
            if (vma)
                zap_page_range(vma, (uintptr_t)page_addr +
                    proc->user_buffer_offset, PAGE_SIZE, NULL);
err_vm_insert_page_failed:
            unmap_kernel_range((unsigned long)page_addr, PAGE_SIZE);
err_map_kernel_failed:
            __free_page(*page);
            *page = NULL;
err_alloc_page_failed:
            ;
        }
err_no_vma:
        if (mm) {
            up_write(&mm->mmap_sem);
            mmput(mm);
        }
        return -ENOMEM;
}
```

通过上述代码不但可以分配物理页面，而且可以用来释放物理页面，这可以通过参数 allocate 来区别，在此只需关注分配物理页面的情况。要分配物理页面的虚拟地址空间范围为（start~end），函数前面的一些检查逻辑不再介绍，此处只需直接看中间的 for 循环，具体实现流程如下所示。

（1）调用 alloc_page()分配一个物理页面，此函数返回一个结构体 page 物理页面描述符，根据这个描述的内容初始化结构体 vm_struct tmp_area。

（2）通过 map_vm_area 将这个物理页面插入到 tmp_area 描述的内核空间。

（3）通过 page_addr + proc->user_buffer_offset 获得进程虚拟空间地址。

（4）通过函数 vm_insert_page()将这个物理页面插入到进程地址空间，参数 vma 表示要插入的进程的地址空间。

中间的 for 循环部分的具体代码如下所示。

```
for (page_addr = start; page_addr < end; page_addr += PAGE_SIZE) {
        int ret;
        struct page **page_array_ptr;
        page = &proc->pages[(page_addr - proc->buffer) / PAGE_SIZE];
```

```
        BUG_ON(*page);
        *page = alloc_page(GFP_KERNEL | __GFP_ZERO);
        if (*page == NULL) {
                printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
                        "for page at %p\n", proc->pid, page_addr);
                goto err_alloc_page_failed;
        }
        tmp_area.addr = page_addr;
        tmp_area.size = PAGE_SIZE + PAGE_SIZE /* guard page? */;
        page_array_ptr = page;
        ret = map_vm_area(&tmp_area, PAGE_KERNEL, &page_array_ptr);
        if (ret) {
                printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
                        "to map page at %p in kernel\n",
                        proc->pid, page_addr);
                goto err_map_kernel_failed;
        }
        user_page_addr =
                (uintptr_t)page_addr + proc->user_buffer_offset;
        ret = vm_insert_page(vma, user_page_addr, page[0]);
        if (ret) {
                printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
                        "to map page at %lx in userspace\n",
                        proc->pid, user_page_addr);
                goto err_vm_insert_page_failed;
        }
        /* vm_insert_page does not seem to increment the refcount */
}
```

再次回到文件 frameworks/base/cmds/servicemanager/service_manager.c 中的 main()函数，接下来需要调用 binder_become_context_manager 来通知 Binder 驱动程序自己是 Binder 机制的上下文管理者，即保护进程。函数 binder_become_context_manager()在文件 frameworks/base/cmds/servicemanager/binder.c 中定义，具体代码如下所示。

```
int binder_become_context_manager(struct binder_state *bs){
    return ioctl(bs->fd, BINDER_SET_CONTEXT_MGR, 0);
}
```

在此通过调用 ioctl 文件操作函数通知 Binder 驱动程序自己是保护进程，命令行是 BINDER_SET_CONTEXT_MGR，并没有任何参数。BINDER_SET_CONTEXT_MGR 定义如下。

```
#define BINDER_SET_CONTEXT_MGR_IOW('b', 7, int)
```

这样就进入到 Binder 驱动程序的函数 binder_ioctl()，在此只关注如下 BINDER_SET_CONTEXT_MGR 命令即可。

```
static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
        int ret;
        struct binder_proc *proc = filp->private_data;
```

```
	struct binder_thread *thread;
	unsigned int size = _IOC_SIZE(cmd);
	void __user *ubuf = (void __user *)arg;
	/*printk(KERN_INFO "binder_ioctl: %d:%d %x %lx\n", proc->pid, current->pid, cmd, arg);*/
	ret = wait_event_interruptible(binder_user_error_wait, binder_stop_on_user_error < 2);
	if (ret)
		return ret;
	mutex_lock(&binder_lock);
	thread = binder_get_thread(proc);
	if (thread == NULL) {
		ret = -ENOMEM;
		goto err;
	}
	switch (cmd) {
	   ...
	case BINDER_SET_CONTEXT_MGR:
		if (binder_context_mgr_node != NULL) {
			printk(KERN_ERR "binder: BINDER_SET_CONTEXT_MGR already set\n");
			ret = -EBUSY;
			goto err;
		}
		if (binder_context_mgr_uid != -1) {
			if (binder_context_mgr_uid != current->cred->euid) {
				printk(KERN_ERR "binder: BINDER_SET_"
					"CONTEXT_MGR bad uid %d != %d\n",
					current->cred->euid,
					binder_context_mgr_uid);
				ret = -EPERM;
				goto err;
			}
		} else
			binder_context_mgr_uid = current->cred->euid;
		binder_context_mgr_node = binder_new_node(proc, NULL, NULL);
		if (binder_context_mgr_node == NULL) {
			ret = -ENOMEM;
			goto err;
		}
		binder_context_mgr_node->local_weak_refs++;
		binder_context_mgr_node->local_strong_refs++;
		binder_context_mgr_node->has_strong_ref = 1;
		binder_context_mgr_node->has_weak_ref = 1;
		break;
	   ...
	default:
		ret = -EINVAL;
		goto err;
	}
	ret = 0;
err:
	if (thread)
		thread->looper &= ~BINDER_LOOPER_STATE_NEED_RETURN;
```

```
        mutex_unlock(&binder_lock);
        wait_event_interruptible(binder_user_error_wait, binder_stop_on_user_error < 2);
        if (ret && ret != -ERESTARTSYS)
                printk(KERN_INFO "binder: %d:%d ioctl %x %lx returned %d\n", proc->pid, current->pid, cmd, arg, ret);
        return ret;
}
```

在分析函数 binder_ioctl()之前，需要先理解如下两个数据结构。

☑　结构体 binder_thread：表示一个线程，这里就是执行 binder_become_context_manager()函数的线程。

```
struct binder_thread {
        struct binder_proc *proc;
        struct rb_node rb_node;
        int pid;
        int looper;
        struct binder_transaction *transaction_stack;
        struct list_head todo;
        uint32_t return_error; /* Write failed, return error code in read buf */
        uint32_t return_error2; /* Write failed, return error code in read */
                /* buffer. Used when sending a reply to a dead process that */
                /* we are also waiting on */
        wait_queue_head_t wait;
        struct binder_stats stats;
};
```

在上述结构体中，proc 表示是这个线程所属的进程。结构体 binder_proc 中成员变量 thread 的类型是 rb_root，表示查询红黑树，把属于这个进程的所有线程都组织起来，结构体 binder_thread 的成员变量 rb_node 就是用来连入这棵红黑树的节点。looper 成员变量表示线程的状态，可以取下面的值。

```
enum {
        BINDER_LOOPER_STATE_REGISTERED     = 0x01,
        BINDER_LOOPER_STATE_ENTERED        = 0x02,
        BINDER_LOOPER_STATE_EXITED         = 0x04,
        BINDER_LOOPER_STATE_INVALID        = 0x08,
        BINDER_LOOPER_STATE_WAITING        = 0x10,
        BINDER_LOOPER_STATE_NEED_RETURN  = 0x20
};
```

另外，transaction_stack 表示线程正在处理的事务，todo 表示发往该线程的数据列表，return_error 和 return_error2 表示操作结果返回码，wait 用来阻塞线程等待某个事件的发生，stats 用来保存一些统计信息，此处暂不详细介绍，遇到这些成员变量时再分析其作用。

☑　数据结构 binder_node：表示一个 Binder 实体，定义如下。

```
struct binder_node {
        int debug_id;
        struct binder_work work;
        union {
                struct rb_node rb_node;
                struct hlist_node dead_node;
```

```
	};
	struct binder_proc *proc;
	struct hlist_head refs;
	int internal_strong_refs;
	int local_weak_refs;
	int local_strong_refs;
	void __user *ptr;
	void __user *cookie;
	unsigned has_strong_ref : 1;
	unsigned pending_strong_ref : 1;
	unsigned has_weak_ref : 1;
	unsigned pending_weak_ref : 1;
	unsigned has_async_transaction : 1;
	unsigned accept_fds : 1;
	int min_priority : 8;
	struct list_head async_todo;
};
```

由此可见，rb_node 和 dead_node 组成了一个联合体，具体来说分为如下两种情形。

➢ 如果这个 Binder 实体还在正常使用，则使用 rb_node 来连入 proc->nodes 所表示的红黑树的节点，这棵红黑树用来组织属于这个进程的所有 Binder 实体。

➢ 如果这个 Binder 实体所属的进程已经销毁，而这个 Binder 实体又被其他进程所引用，则它通过 dead_node 进入到一个哈希表中存放。proc 成员变量就是表示这个 Binder 实例所属的进程。

refs 成员变量把所有引用了该 Binder 实体的 Binder 引用连接起来构成一个链表。internal_strong_refs、local_weak_refs 和 local_strong_refs 表示这个 Binder 实体的引用计数。ptr 和 cookie 成员变量分别表示这个 Binder 实体在用户空间的地址以及附加数据。其余的成员变量不再描述了，遇到时再分析。

接下来回到函数 binder_ioctl()中，首先是通过 filp->private_data 获得 proc 变量，此处的函数 binder_mmap()是一样的，然后通过函数 binder_get_thread()获得线程信息，此函数会把当前线程 current 的 pid 作为键值，在进程 proc->threads 表示的红黑树中进行查找，看是否已经为当前线程创建了 binder_thread 信息。在这个场景下，由于当前线程是第一次进到这里，所以肯定找不到，即*p == NULL 成立，于是，就为当前线程创建一个线程上下文信息结构体 binder_thread，初始化相应成员变量，并插入到 proc->threads 所表示的红黑树中，下次要使用时就可以从 proc 中找到了。注意，这里的 thread->looper = BINDER_LOOPER_STATE_NEED_RETURN。函数 binder_get_thread()的具体代码如下所示。

```
static struct binder_thread *binder_get_thread(struct binder_proc *proc)
{
	struct binder_thread *thread = NULL;
	struct rb_node *parent = NULL;
	struct rb_node **p = &proc->threads.rb_node;

	while (*p) {
		parent = *p;
		thread = rb_entry(parent, struct binder_thread, rb_node);

		if (current->pid < thread->pid)
			p = &(*p)->rb_left;
		else if (current->pid > thread->pid)
```

```
            p = &(*p)->rb_right;
        else
            break;
    }
    if (*p == NULL) {
        thread = kzalloc(sizeof(*thread), GFP_KERNEL);
        if (thread == NULL)
            return NULL;
        binder_stats.obj_created[BINDER_STAT_THREAD]++;
        thread->proc = proc;
        thread->pid = current->pid;
        init_waitqueue_head(&thread->wait);
        INIT_LIST_HEAD(&thread->todo);
        rb_link_node(&thread->rb_node, parent, p);
        rb_insert_color(&thread->rb_node, &proc->threads);
        thread->looper |= BINDER_LOOPER_STATE_NEED_RETURN;
        thread->return_error = BR_OK;
        thread->return_error2 = BR_OK;
    }
    return thread;
}
```

再回到函数 binder_ioctl()中，接下来会有两个全局变量 binder_context_mgr_node 和 binder_context_mgr_uid，定义如下。

```
static struct binder_node *binder_context_mgr_node;
static uid_t binder_context_mgr_uid = -1;
```

其中，binder_context_mgr_node 用来表示 Service Manager 实体，binder_context_mgr_uid 表示 Service Manager 保护进程的 uid。在这个场景下，由于当前线程是第一次进到这里，所以 binder_context_mgr_node 为 NULL，binder_context_mgr_uid 为-1，于是初始化 binder_context_mgr_uid 为 current->cred->euid，这样当前线程就成为 Binder 机制的保护进程了，并且通过 binder_new_node 为 Service Manager 创建 Binder 实体。

```
static struct binder_node *
binder_new_node(struct binder_proc *proc, void __user *ptr, void __user *cookie)
{
    struct rb_node **p = &proc->nodes.rb_node;
    struct rb_node *parent = NULL;
    struct binder_node *node;
    while (*p) {
        parent = *p;
        node = rb_entry(parent, struct binder_node, rb_node);
        if (ptr < node->ptr)
            p = &(*p)->rb_left;
        else if (ptr > node->ptr)
            p = &(*p)->rb_right;
        else
            return NULL;
    }
```

```
    node = kzalloc(sizeof(*node), GFP_KERNEL);
    if (node == NULL)
        return NULL;
    binder_stats.obj_created[BINDER_STAT_NODE]++;
    rb_link_node(&node->rb_node, parent, p);
    rb_insert_color(&node->rb_node, &proc->nodes);
    node->debug_id = ++binder_last_id;
    node->proc = proc;
    node->ptr = ptr;
    node->cookie = cookie;
    node->work.type = BINDER_WORK_NODE;
    INIT_LIST_HEAD(&node->work.entry);
    INIT_LIST_HEAD(&node->async_todo);
    if (binder_debug_mask & BINDER_DEBUG_INTERNAL_REFS)
        printk(KERN_INFO "binder: %d:%d node %d u%p c%p created\n",
               proc->pid, current->pid, node->debug_id,
               node->ptr, node->cookie);
    return node;
}
```

在这里传进来的 ptr 和 cookie 都为 NULL。上述函数会首先检查 proc->nodes 红黑树中是否已经存在以 ptr 为键值的 node，如果已经存在则返回 NULL。在这个场景下，由于当前线程是第一次进入到这里，所以肯定不存在，于是就新建了一个 ptr 为 NULL 的 binder_node，并且初始化其他成员变量，插入到 proc->nodes 红黑树中去。

当 binder_new_node 返回到函数 binder_ioctl()后，会把新建的 binder_node 指针保存在 binder_context_mgr_node 中，然后又初始化 binder_context_mgr_node 的引用计数值。这样执行 BINDER_SET_CONTEXT_MGR 命令完毕，在函数 binder_ioctl()返回之前执行下面的语句。

```
if (thread)
        thread->looper &= ~BINDER_LOOPER_STATE_NEED_RETURN;
```

再次回到文件 frameworks/base/cmds/servicemanager/service_manager.c 中的 main()函数，接下来需要调用函数 binder_loop()进入循环，等待 Client 发送请求。函数 binder_loop()在文件 frameworks/base/cmds/servicemanager/binder.c 中定义，首先通过函数 binder_write()执行 BC_ENTER_LOOPER 命令，告诉 Binder 驱动程序 Service Manager 马上要进入循环。函数 binder_loop()的具体实现代码如下所示。

```
void binder_loop(struct binder_state *bs, binder_handler func)
{
    int res;
    struct binder_write_read bwr;
    unsigned readbuf[32];
    bwr.write_size = 0;
    bwr.write_consumed = 0;
    bwr.write_buffer = 0;

    readbuf[0] = BC_ENTER_LOOPER;
    binder_write(bs, readbuf, sizeof(unsigned));
    for (;;) {
        bwr.read_size = sizeof(readbuf);
```

```
            bwr.read_consumed = 0;
            bwr.read_buffer = (unsigned) readbuf;
            res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);
            if (res < 0) {
                  LOGE("binder_loop: ioctl failed (%s)\n", strerror(errno));
                  break;
            }
            res = binder_parse(bs, 0, readbuf, bwr.read_consumed, func);
            if (res == 0) {
                  LOGE("binder_loop: unexpected reply?!\n");
                  break;
            }
            if (res < 0) {
                  LOGE("binder_loop: io error %d %s\n", res, strerror(errno));
                  break;
            }
      }
}
```

在此还需要理解设备文件/dev/binder 操作函数 ioctl 的操作码 BINDER_WRITE_READ，首先看其定义。

```
#define BINDER_WRITE_READ _IOWR('b', 1, struct binder_write_read)
```

此 io 操作码有一个形式为 struct binder_write_read 的参数。

```
struct binder_write_read {
      signed long     write_size;       /* bytes to write */
      signed long     write_consumed;       /* bytes consumed by driver */
      unsigned long   write_buffer;
      signed long     read_size;/* bytes to read */
      signed long     read_consumed;       /* bytes consumed by driver */
      unsigned long   read_buffer;
};
```

用户空间程序和 Binder 驱动程序交互时，大多数是通过 BINDER_WRITE_READ 命令实现的，write_buffer 和 read_buffer 所指向的数据结构还指定了具体要执行的操作，write_buffer 和 read_buffer 所指向的结构体是 binder_transaction_data，定义此结构体的代码如下所示。

```
struct binder_transaction_data {
      /* The first two are only used for bcTRANSACTION and brTRANSACTION,
       * identifying the target and contents of the transaction
       */
      union {
            size_t handle;          /* target descriptor of command transaction */
            void *ptr;              /* target descriptor of return transaction */
      } target;
      Void *cookie;                 /* target object cookie */
      unsigned int code;            /* transaction command */

      /* General information about the transaction. */
```

```
        unsigned int flags;
        pid_t sender_pid;
        uid_t sender_euid;
        size_t data_size;           /* number of bytes of data */
        size_t offsets_size;        /* number of bytes of offsets */

        /* If this transaction is inline, the data immediately
         * follows here; otherwise, it ends with a pointer to
         * the data buffer
         */
        union {
                struct {
                        /* transaction data */
                        const void *buffer;
                        /* offsets from buffer to flat_binder_object structs */
                        const void *offsets;
                } ptr;
                uint8_t     buf[8];
        } data;
};
```

到此为止，已从源代码一步一步地分析完 Service Manager 是如何成为 Android 进程间通信（IPC）机制 Binder 保护进程的了。下面简要总结 Service Manager 成为 Android 进程间通信（IPC）机制 Binder 保护进程的过程。

（1）打开/dev/binder 文件。

```
open("/dev/binder", O_RDWR);
```

（2）建立 128K 内存映射。

```
mmap(NULL, mapsize, PROT_READ, MAP_PRIVATE, bs->fd, 0);
```

（3）通知 Binder 驱动程序它是保护进程。

```
binder_become_context_manager(bs);
```

（4）进入循环等待请求的到来。

```
binder_loop(bs, svcmgr_handler);
```

在这个过程中，在 Binder 驱动程序中建立了一个 struct binder_proc 结构、一个 struct binder_thread 结构和一个 struct binder_node 结构，这样，Service Manager 就在 Android 系统的进程间通信机制 Binder 担负起保护进程的职责了。

3.1.3 Service Manager 服务

在 Android 5.0 中，Service Manager 在 Binder 机制中既充当保护进程的角色，同时也充当着 Server 角色，但是它又与一般的 Server 不一样。对于普通的 Server 来说，Client 如果想要获得 Server 的远程接口，必须通过 Service Manager 远程接口提供的 getService 接口获得，这本身就是一个使用 Binder 机

制来进行进程间通信的过程。而对于 Service Manager 这个 Server 来说，Client 如果想要获得 Service Manager 远程接口，却不必通过进程间通信机制来获得，因为 Service Manager 远程接口是一个特殊的 Binder 引用，其引用句柄一定是 0。

在 Android 5.0 中，获取 Service Manager 远程接口的函数是 defaultServiceManager()，此函数在文件 frameworks/base/include/binder/IServiceManager.h 中声明，具体代码如下所示。

```
sp<IServiceManager> defaultServiceManager();
```

函数 defaultServiceManager()在文件 frameworks/base/libs/binder/IServiceManager.cpp 中实现，具体代码如下所示。

```
sp<IServiceManager> defaultServiceManager()
{
    if (gDefaultServiceManager != NULL) return gDefaultServiceManager;
    {
        AutoMutex _l(gDefaultServiceManagerLock);
        if (gDefaultServiceManager == NULL) {
            gDefaultServiceManager = interface_cast<IServiceManager>(
                ProcessState::self()->getContextObject(NULL));
        }
    }
    return gDefaultServiceManager;
}
```

其中 gDefaultServiceManagerLock 和 gDefaultServiceManager 是全局变量，在文件 frameworks/base/libs/binder/Static.cpp 中定义，具体代码如下所示。

```
Mutex gDefaultServiceManagerLock;
sp<IServiceManager> gDefaultServiceManager;
```

从上述函数可以看出，gDefaultServiceManager 是单例模式，在调用函数 defaultServiceManager()时，如果已经创建了 gDefaultServiceManager 则直接返回，否则通过 interface_cast<IServiceManager>(ProcessState::self()->getContextObject(NULL))创建一个，并保存在全局变量 gDefaultServiceManager 中。

在 Binder 机制中，类 BpServiceManager 继承了类 BpInterface<IServiceManager>，BpInterface 是一个模板类，在文件 frameworks/base/include/binder/IInterface.h 中定义，具体代码如下所示。

```
template<typename INTERFACE>
class BpInterface : public INTERFACE, public BpRefBase {
public:
    BpInterface(const sp<IBinder>& remote);
protected:
    virtual IBinder* onAsBinder();
};
```

类 IServiceManager 继承了类 IInterface，而类 IInterface 和类 BpRefBase 又分别继承了类 RefBase。

下面是创建 Service Manager 远程接口的主要语句，首先会调用函数 ProcessState::self()，此函数是 ProcessState 的静态成员函数，功能是返回一个全局唯一的 ProcessState 实例变量，其实这就是单例模式，此变量名为 gProcess。如果未创建 gProcess 则执行创建操作。在 ProcessState 的构造函数中，通过文件操

作函数 open()打开设备文件/dev/binder，并且返回的设备文件描述符保存在成员变量 mDriverFD 中。

```
gDefaultServiceManager = interface_cast<IServiceManager>(
        ProcessState::self()->getContextObject(NULL));
```

接着调用函数 gProcess->getContextObject()获得一个句柄值为 0 的 Binder 引用 BpBinder。再来看函数 interface_cast<IServiceManager>的具体实现，此模板函数在文件 framework/base/include/binder/IInterface.h 中定义，具体实现代码如下所示。

```
template<typename INTERFACE>
inline sp<INTERFACE> interface_cast(const sp<IBinder>& obj) {
        return INTERFACE::asInterface(obj);
}
```

在上述代码中，INTERFACE 是 IServiceManager，调用了函数 IServiceManager::asInterface()。函数 IServiceManager::asInterface()是通过 DECLARE_META_INTERFACE(ServiceManager)宏在类 IServiceManager 中声明的，位于文件 framework/base/include/binder/IServiceManager.h 中，展开后的代码如下所示。

```
#define DECLARE_META_INTERFACE(ServiceManager)                          \
        static const android::String16 descriptor;                 \
        static android::sp<IServiceManager> asInterface(           \
        const android::sp<android::IBinder>& obj);                 \
        virtual const android::String16& getInterfaceDescriptor() const;      \
        IServiceManager();                                          \
        virtual ~IServiceManager();
```

IServiceManager::asInterface 是通过宏 IMPLEMENT_META_INTERFACE(ServiceManager, "android.os.IServiceManager")定义的，位于文件 framework/base/libs/binder/IServiceManager.cpp 中，展开后的代码如下所示。

```
#define IMPLEMENT_META_INTERFACE(ServiceManager, "android.os.IServiceManager")   \
        const android::String16 IServiceManager::descriptor("android.os.IServiceManager");      \
        const android::String16&                                    \
        IServiceManager::getInterfaceDescriptor() const {                    \
        return IServiceManager::descriptor;                                   \
        }                                                                       \
        android::sp<IServiceManager> IServiceManager::asInterface(             \
        const android::sp<android::IBinder>& obj)                             \
        {                                                                       \
        android::sp<IServiceManager> intr;                                     \
        if (obj != NULL) {                                                     \
        intr = static_cast<IServiceManager*>(                                  \
        obj->queryLocalInterface(                                              \
        IServiceManager::descriptor).get());                                  \
        if (intr == NULL) {                                                    \
        intr = new BpServiceManager(obj);                                      \
        }                                                                       \
        }                                                                       \
        return intr;                                                           \
        }                                                                       \
```

```
    IServiceManager::IServiceManager() { }                                    \
    IServiceManager::~IServiceManager() { }
```

IServiceManager::asInterface 的具体实现代码如下所示。

```
android::sp<IServiceManager> IServiceManager::asInterface(const android::sp<android::IBinder>& obj)
{
    android::sp<IServiceManager> intr;

    if (obj != NULL) {
        intr = static_cast<IServiceManager*>(
                    obj->queryLocalInterface(IServiceManager::descriptor).get());

        if (intr == NULL) {
            intr = new BpServiceManager(obj);
        }
    }
    return intr;
}
```

此处传进来的参数 obj 就是刚才创建的 new BpBinder(0)，类 BpBinder 中的成员函数 queryLocalInterface()继承自基类 IBinder，函数 IBinder::queryLocalInterface()位于文件 framework/base/libs/binder/Binder.cpp 中，具体实现代码如下所示。

```
sp<IInterface>   IBinder::queryLocalInterface(const String16& descriptor)
{
   return NULL;
}
```

由此可见，在函数 IServiceManager::asInterface()中会调用下面的语句。

```
intr = new BpServiceManager(obj);
```

即：

```
intr = new BpServiceManager(new BpBinder(0));
```

创建的 Service Manager 远程接口本质上是一个 BpServiceManager，包含了一个句柄值为 0 的 Binder 引用。

3.2　分析匿名共享内存子系统

Android 系统提供了独特的匿名共享内存子系统 Ashmem（Anonymous Shared Memory），以驱动程序的形式实现在内核空间中。本节将详细分析 Android 匿名共享内存子系统 Ashmem 的基本源码。

3.2.1　Ashmem 系统基础

在 Android 5.0 系统中，Ashmem 具有如下两个特点。

☑ 能够辅助内存管理系统有效地管理不再使用的内存块。

☑ 通过 Binder 进程间通信机制来实现进程间的内存共享。

对于 Android 系统的匿名共享内存子系统来说，其主体是以驱动程序的形式实现在内核空间的，同时，在系统运行时库层和应用程序框架层提供了访问接口。其中在系统运行时库层提供了 C/C++调用接口，而在应用程序框架层提供了 Java 调用接口。在此将直接通过应用程序框架层提供的 Java 调用接口来说明匿名共享内存子系统 Ashmem 的使用方法，毕竟在 Android 开发应用程序时，是基于 Java 语言的。其实应用程序框架层的 Java 调用接口是通过 JNI 方法来调用系统运行时库层的 C/C++调用接口，最后进入到内核空间的 Ashmem 驱动程序中的。

Android 系统的匿名共享内存 Ashmem 驱动程序利用了 Linux 的共享内存子系统导出的接口来实现自己的功能。在 Android 系统匿名共享内存系统中，其核心功能是实现创建（open）、映射（mmap）、读写（read/write）以及锁定和解锁（pin/unpin）。

3.2.2 基础数据结构

在 Ashmem 驱动程序中，用到了 ashmem_area、ashmem_range 和 ashmem_pin 这 3 个结构体。其中前两个结构体在文件 kernel/goldfish/mm/ashmem.c 中定义，实现代码如下所示。

```
struct ashmem_area {
	char name[ASHMEM_FULL_NAME_LEN];	/*匿名共享内存的名称*/
	struct list_head unpinned_list;		/*解锁内存列表*/
	struct file *file;			/*指向临时文件系统 tmpfs 中的一个文件*/
	size_t size;				/*文件大小*/
	unsigned long prot_mask;		/*匿名共享内存的访问保护位*/
};
struct ashmem_range {
	struct list_head lru;			/*最近最少使用的列表*/
	struct list_head unpinned;
	struct ashmem_area *asma;
	size_t pgstart;				/*处于解锁状态内存的开始地址*/
	size_t pgend;				/*处于解锁状态内存的结束地址*/
	unsigned int purged;			/*解锁内存是否被收回*/
};
```

结构体 ashmem_area 用于表示一块匿名共享内存单元，结构体 ashmem_range 用于表示处于解锁状态的内存。

结构体 ashmem_pin 用于表示被锁定或被解锁的内存，在文件 kernel/goldfish/include/linux/ashmem.h 中定义，具体代码如下所示。

```
struct ashmem_pin {
	__u32 offset;				/*这块内存的偏移值*/
	__u32 len;				/*这块内存的大小*/
};
```

结构体 ashmem_fops 定义了 dev/ashmem 的操作方法列表，具体代码如下所示。

```
static struct file_operations ashmem_fops = {
	.owner = THIS_MODULE,
```

```
            .open = ashmem_open,
            .release = ashmem_release,
            .mmap = ashmem_mmap,
            .unlocked_ioctl = ashmem_ioctl,
            .compat_ioctl = ashmem_ioctl,
};
```

3.2.3　初始化处理

通过 Ashmem 驱动初始化函数可以获取如下两点信息。

☑　Ashmem 给用户空间暴露了什么接口，即创建了什么样的设备文件。

☑　Ashmem 提供了什么函数来操作这个设备文件。

Ashmem 驱动程序在文件 kernel/common/mm/ashmem.c 中实现，其中函数 ashmem_init()实现模块初始化处理，主要实现代码如下所示。

```
static struct miscdevice ashmem_misc = {
        .minor = MISC_DYNAMIC_MINOR,
        .name = "ashmem",
        .fops = &ashmem_fops,
};
static int __init ashmem_init(void)
{
        int ret;
        ...
        ret = misc_register(&ashmem_misc);
        if (unlikely(ret)) {
                printk(KERN_ERR "ashmem: failed to register misc device!\n");
                return ret;
        }
        ...
        return 0;
}
```

在上述代码中，在加载 Ashmem 驱动程序时会创建一个设备文件/dev/ashmem，这是一个 misc 类型的设备。通过函数 misc_register()来注册 misc 设备，调用这个函数后会在/dev 目录下生成一个 ashmem 设备文件。在设备文件中一共提供了 open、mmap、release 和 ioctl 共 4 种操作，此处并没有 read 和 write 操作，原因是读写共享内存的方法是通过内存映射地址来进行的，通过 mmap 系统调用将这个设备文件映射到进程地址空间中。与此同时，直接对内存进行了读写操作，所以不需要通过 read 和 write 方式进行文件操作。

匿名共享内存创建功能是在文件 frameworks/base/core/java/android/os/MemoryFile.java 中实现的，此文件调用了类 MemoryFile 的构造函数，MemoryFile 的构造函数调用了 JNI 函数 native_open，这样便创建了匿名内存共享文件。JNI 方法 native_open()在文件 frameworks/base/core/jni/adroid_os_MemoryFile.cpp 中实现，具体代码如下所示。

```
static jobject android_os_MemoryFile_open(JNIEnv* env, jobject clazz, jstring name, jint length)
{
            const char* namestr = (name ? env->GetStringUTFChars(name, NULL) : NULL);
```

```
        int result = ashmem_create_region(namestr, length);

        if (name)
            env->ReleaseStringUTFChars(name, namestr);

        if (result < 0) {
            jniThrowException(env, "java/io/IOException", "ashmem_create_region failed");
            return NULL;
        }

        return jniCreateFileDescriptor(env, result);
}
```

函数 native_open()通过运行时库提供的接口 ashmem_create_region()创建匿名共享内存，这个接口在文件 system/core/libcutils/ashmem-dev.c 中实现，具体代码如下所示。

```
int ashmem_create_region(const char *name, size_t size)
{
    int fd, ret;

    fd = open(ASHMEM_DEVICE, O_RDWR);
    if (fd < 0)
        return fd;

    if (name) {
        char buf[ASHMEM_NAME_LEN];

        strlcpy(buf, name, sizeof(buf));
        ret = ioctl(fd, ASHMEM_SET_NAME, buf);
        if (ret < 0)
            goto error;
    }

    ret = ioctl(fd, ASHMEM_SET_SIZE, size);
    if (ret < 0)
        goto error;

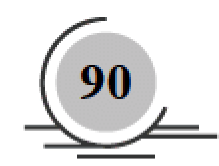
    return fd;

error:
    close(fd);
    return ret;
}
```

在上述代码中，通过执行 3 个文件操作系统调用的方式和 Ashmem 驱动程序进行交互。通过 open 操作打开设备文件 ASHMEM_DEVICE，通过 ioctl 操作设置匿名共享内存的名称和大小。

3.2.4 打开匿名共享内存设备文件

open 进入内核后会调用函数 ashmem_open()打开匿名共享内存设备文件，此函数能够为程序创建



一个 ashmem_area 结构体，具体实现代码如下所示。

```
static int ashmem_open(struct inode *inode, struct file *file)
{
	struct ashmem_area *asma;
	int ret;
	ret = nonseekable_open(inode, file);
	if (unlikely(ret))
		return ret;
	asma = kmem_cache_zalloc(ashmem_area_cachep, GFP_KERNEL);
	if (unlikely(!asma))
		return -ENOMEM;
	INIT_LIST_HEAD(&asma->unpinned_list);
	memcpy(asma->name, ASHMEM_NAME_PREFIX, ASHMEM_NAME_PREFIX_LEN);
	asma->prot_mask = PROT_MASK;
	file->private_data = asma;
	return 0;
}
```

上述代码的执行流程如下所示。

☑ 通过函数 nonseekable_open()设置这个文件不可以执行定位操作，即不可执行 seek 文件操作。

☑ 通过函数 kmem_cache_zalloc()在刚创建的 slab 缓冲区 ashmem_area_cachep 中创建一个 ashmem_area 结构体，并将创建的结构体保存在本地变量 asma 中。

☑ 初始化变量 asma 的其他域，其中，域 name 初始化为宏 ASHMEM_NAME_PREFIX 和宏 ASHMEM_NAME_PREFIX_LEN 的定义代码如下。

```
#define ASHMEM_NAME_PREFIX "dev/ashmem/"
#define ASHMEM_NAME_PREFIX_LEN (sizeof(ASHMEM_NAME_PREFIX) - 1)
```

☑ 将结构 ashmem_area 保存在打开的文件结构体的 private_data 域中，此时通过使用 Ashmem 驱动程序，可以在其他模块通过 private_data 域来取回这个 ashmem_area 结构。

在函数 ashmem_create_region()中调用了两次 ioctl 文件操作，功能是设置新建匿名共享内存的名字和大小。在文件 kernel/comon/mm/include/ashmem.h 中，ASHMEM_SET_NAME 和 ASHMEM_SET_SIZE 分别表示新建内存的名字和大小，具体定义代码如下所示。

```
#define ASHMEM_NAME_LEN 256
#define __ASHMEMIOC 0x77
#define ASHMEM_SET_NAME _IOW(__ASHMEMIOC, 1, char[ASHMEM_NAME_LEN])
#define ASHMEM_SET_SIZE _IOW(__ASHMEMIOC, 3, size_t)
```

其中，ASHMEM_SET_NAME 的 ioctl 调用会进入到 Ashmem 驱动程序函数 ashmem_ioctl()中，此函数能够将从用户空间传进来的匿名共享内存的大小值保存在对应的 asma->size 域中。函数 ashmem_ioctl()的实现代码如下所示。

```
static long ashmem_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
		struct ashmem_area *asma = file->private_data;
		long ret = -ENOTTY;
```

```
	switch (cmd) {
	case ASHMEM_SET_NAME:
		ret = set_name(asma, (void __user *) arg);
		break;
	case ASHMEM_GET_NAME:
		ret = get_name(asma, (void __user *) arg);
		break;
	case ASHMEM_SET_SIZE:
		ret = -EINVAL;
		if (!asma->file) {
			ret = 0;
			asma->size = (size_t) arg;
		}
		break;
	case ASHMEM_GET_SIZE:
		ret = asma->size;
		break;
	case ASHMEM_SET_PROT_MASK:
		ret = set_prot_mask(asma, arg);
		break;
	case ASHMEM_GET_PROT_MASK:
		ret = asma->prot_mask;
		break;
	case ASHMEM_PIN:
	case ASHMEM_UNPIN:
	case ASHMEM_GET_PIN_STATUS:
		ret = ashmem_pin_unpin(asma, cmd, (void __user *) arg);
		break;
	case ASHMEM_PURGE_ALL_CACHES:
		ret = -EPERM;
		if (capable(CAP_SYS_ADMIN)) {
			ret = ashmem_shrink(0, GFP_KERNEL);
			ashmem_shrink(ret, GFP_KERNEL);
		}
		break;
	}
	return ret;
}
```

上述代码主要完成如下两个功能。

☑ struct ashmem_area *asma = file->private_data：获取描述将要改名的匿名共享内存 asma。

☑ ret = set_name(asma, (void __user *) arg)：调用函数 set_name()修改匿名共享内存的名称。

函数 set_name()也是在文件 kernel/goldfish/mm/ashmem.c 中实现的，功能是把用户空间传进来的匿名共享内存的名字设置到 asma->name 域中。函数 set_name()的具体实现代码如下所示。

```
static int set_name(struct ashmem_area *asma, void __user *name)
{
	int ret = 0;
	mutex_lock(&ashmem_mutex);
	/* cannot change an existing mapping's name */
```

```
        if (unlikely(asma->file)) {
                ret = -EINVAL;
                goto out;
        }
        if (unlikely(copy_from_user(asma->name + ASHMEM_NAME_PREFIX_LEN,
                                        name, ASHMEM_NAME_LEN)))
                ret = -EFAULT;
        asma->name[ASHMEM_FULL_NAME_LEN-1] = '\0';
out:
        mutex_unlock(&ashmem_mutex);
        return ret;
}
```

到此为止，创建匿名共享内存的过程就全部介绍完毕了。

3.2.5　实现内存映射

Ashmem 驱动程序并不提供文件的 read 操作和 write 操作，如果进程要访问这个共享内存，则必须将这个设备文件映射到自己的进程空间中，然后才能进行内存访问。在类 MemoryFile 的构造函数中，创建匿名共享内存后需要把匿名共享内存设备文件映射到进程空间。映射功能是通过调用 JNI 方法 native_mmap()实现的，此 JNI 方法在文件 frameworks/base/core/jni/adroid_os_MemoryFile.cpp 中实现，具体代码如下所示。

```
static jint android_os_MemoryFile_mmap(JNIEnv* env, jobject clazz, jobject fileDescriptor,
        jint length, jint prot)
{
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    jint result = (jint)mmap(NULL, length, prot, MAP_SHARED, fd, 0);
    if (!result)
        jniThrowException(env, "java/io/IOException", "mmap failed");
    return result;
}
```

上述代码中，在 open 匿名设备文件/dev/ashmem 中获得文件描述符 fd。有了这个文件描述符后，就可以直接通过函数 mmap()执行内存映射操作了。当调用函数 mmap()打开映射到进程的地址空间时，会立即执行 Ashmem 中的函数 ashmem_mmap()。函数 ashmem_mmap()的功能是调用 Linux 内核中的函数 shmem_file_setup()在临时文件系统 tmpfs 中创建一个临时文件，这个临时文件与 Ashmem 驱动程序创建的匿名共享内存对应。函数 ashmem_mmap()在文件 kernel/goldfish/mm/ashmem.c 中定义，具体实现代码如下所示。

```
static int ashmem_mmap(struct file *file, struct vm_area_struct *vma)
{
        struct ashmem_area *asma = file->private_data;
        int ret = 0;
        mutex_lock(&ashmem_mutex);
        /* user needs to SET_SIZE before mapping */
        if (unlikely(!asma->size)) {
                ret = -EINVAL;
```

```
                goto out;
        }
        /* requested protection bits must match our allowed protection mask */
        if (unlikely((vma->vm_flags & ~asma->prot_mask) & PROT_MASK)) {
                ret = -EPERM;
                goto out;
        }
        if (!asma->file) {
                char *name = ASHMEM_NAME_DEF;
                struct file *vmfile;
                if (asma->name[ASHMEM_NAME_PREFIX_LEN] != '\0')
                        name = asma->name;
                /* ... and allocate the backing shmem file */
                vmfile = shmem_file_setup(name, asma->size, vma->vm_flags);
                if (unlikely(IS_ERR(vmfile))) {
                        ret = PTR_ERR(vmfile);
                        goto out;
                }
                asma->file = vmfile;
        }
        get_file(asma->file);
        if (vma->vm_flags & VM_SHARED)
                shmem_set_file(vma, asma->file);
        else {
                if (vma->vm_file)
                        fput(vma->vm_file);
                vma->vm_file = asma->file;
        }
        vma->vm_flags |= VM_CAN_NONLINEAR;
out:
        mutex_unlock(&ashmem_mutex);
        return ret;
}
```

在上述代码中，检查了虚拟内存 vma 是否允许在不同进程之间实现共享。如果允许则调用函数 shmem_set_file()来设置其映射文件和内存操作方法表。

3.2.6 实现读/写操作

从类 MemoryFile 中可以获得读/写操作的过程，对应的代码如下所示。

```
private static native int native_read(FileDescriptor fd, int address, byte[] buffer,
int srcOffset, int destOffset, int count, boolean isUnpinned) throws IOException;
private static native void native_write(FileDescriptor fd, int address, byte[] buffer,
int srcOffset, int destOffset, int count, boolean isUnpinned) throws IOException;
private FileDescriptor mFD;
private int mAddress;
private int mLength;
private boolean mAllowPurging = false;
public int readBytes(byte[] buffer, int srcOffset, int destOffset, int count)
```

```
    throws IOException {
        if (isDeactivated()) {
            throw new IOException("Can't read from deactivated memory file.");
        }
        if (destOffset < 0 || destOffset > buffer.length || count < 0
            || count > buffer.length - destOffset
            || srcOffset < 0 || srcOffset > mLength
            || count > mLength - srcOffset) {
                throw new IndexOutOfBoundsException();
        }
        return native_read(mFD, mAddress, buffer, srcOffset, destOffset, count, mAllowPurging);
    }
    public void writeBytes(byte[] buffer, int srcOffset, int destOffset, int count)
        throws IOException {
            if (isDeactivated()) {
                throw new IOException("Can't write to deactivated memory file.");
            }
            if (srcOffset < 0 || srcOffset > buffer.length || count < 0
                || count > buffer.length - srcOffset
                || destOffset < 0 || destOffset > mLength
                || count > mLength - destOffset) {
                    throw new IndexOutOfBoundsException();
            }
            native_write(mFD, mAddress, buffer, srcOffset, destOffset, count, mAllowPurging);
}
```

通过对上述代码的分析可知，是通过调用 JNI 方法实现读写匿名共享内存操作功能的。读操作的 JNI 接口是 native_read()，即 android_os_MemoryFile_read，写操作的 JNI 接口是 native_write()，即 android_os_MemoryFile_write。这两个方法都在文件 frameworks/base/core/jni/adroid_os_MemoryFile.cpp 中定义，具体实现代码如下所示。

```
static jint android_os_MemoryFile_read(JNIEnv* env, jobject clazz,
        jobject fileDescriptor, jint address, jbyteArray buffer, jint srcOffset, jint destOffset,
        jint count, jboolean unpinned)
{
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    if (unpinned && ashmem_pin_region(fd, 0, 0) == ASHMEM_WAS_PURGED) {
        ashmem_unpin_region(fd, 0, 0);
        jniThrowException(env, "java/io/IOException", "ashmem region was purged");
        return -1;
    }

    env->SetByteArrayRegion(buffer, destOffset, count, (const jbyte *)address + srcOffset);

    if (unpinned) {
        ashmem_unpin_region(fd, 0, 0);
    }
    return count;
}

static jint android_os_MemoryFile_write(JNIEnv* env, jobject clazz,
```

```
        jobject fileDescriptor, jint address, jbyteArray buffer, jint srcOffset, jint destOffset,
        jint count, jboolean unpinned)
{
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    if (unpinned && ashmem_pin_region(fd, 0, 0) == ASHMEM_WAS_PURGED) {
        ashmem_unpin_region(fd, 0, 0);
        jniThrowException(env, "java/io/IOException", "ashmem region was purged");
        return -1;
    }
    env->GetByteArrayRegion(buffer, srcOffset, count, (jbyte *)address + destOffset);
    if (unpinned) {
        ashmem_unpin_region(fd, 0, 0);
    }
    return count;
}
```

在上述代码中，函数 ashmem_pin_region()和 ashmem_unpin_region()用于为系统运行时库提供接口，功能是执行匿名共享内存的锁定和解锁操作。这样便能够通知 Ashmem 驱动程序哪些内存块是正在使用的，需要锁定，哪些不需要使用，可以解锁。这两个函数在文件 system/core/libcutils/ashmem- dev.c 中定义，具体实现代码如下所示。

```
int ashmem_pin_region(int fd, size_t offset, size_t len)
{
    struct ashmem_pin pin = { offset, len };
    return ioctl(fd, ASHMEM_PIN, &pin);
}
int ashmem_unpin_region(int fd, size_t offset, size_t len)
{
    struct ashmem_pin pin = { offset, len };
    return ioctl(fd, ASHMEM_UNPIN, &pin);
}
```

经过上述操作之后，Ashmem 驱动程序就可以在整个内存管理系统中管理内存了。

3.2.7 实现锁定和解锁

在 Android 系统中，通过如下两个 ioctl 操作实现匿名共享内存的锁定和解锁操作。

☑ ASHMEM_PIN

☑ ASHMEM_UNPIN

ASHMEM_PIN 和 ASHMEM_UNPIN 在文件 kernel/common/include/linux/ashmem.h 中定义，对应代码如下所示。

```
#define __ASHMEMIOC 0x77
#define ASHMEM_PIN _IOW(__ASHMEMIOC, 7, struct ashmem_pin)
#define ASHMEM_UNPIN _IOW(__ASHMEMIOC, 8, struct ashmem_pin)
struct ashmem_pin {
    __u32 offset;   /*offset into region, in bytes, page-aligned*/
    __u32 len;      /*length forward from offset, in bytes, page-aligned*/
};
```

再看函数 ashmem_ioctl()，在其实现代码中，与 ASHMEM_PIN 和 ASHMEM_UNPIN 这两个操作相关的代码如下所示。

```
static long ashmem_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
	struct ashmem_area *asma = file->private_data;
	long ret = -ENOTTY;
	switch (cmd) {
	...
	case ASHMEM_PIN:
	case ASHMEM_UNPIN:
		ret = ashmem_pin_unpin(asma, cmd, (void __user *) arg);
		break;
	...
	}
	return ret;
}
```

在上述代码中，调用函数 ashmem_pin_unpin()处理控制命令 ASHMEM_PIN 和 ASHMEM_UNPIN。函数 ashmem_pin_unpin()的实现流程如下所示。

☑ 获取传递到用户空间的参数，并将获取到的值保存在本地变量 pin 中。这是一个 struct ashmem_pin 类型的结构体类型，其中包括了要 pin/unpin 的内存块的起始地址和大小。
☑ 因为起始地址和大小的单位都是字节，所以通过转换处理为以页面为单位并保存在本地变量 pgstart 和 pgend 中。
☑ 不但对参数进行安全性检查，并且确保只要从用户空间传进来的内存块的大小值为 0，就认为是要 pin/unpin 整个匿名共享内存。
☑ 判断当前要执行操作的类别，根据 ASHMEM_PIN 操作和 ASHMEM_UNPIN 操作分别执行 ashmem_pin 和 ashmem_unpin。
☑ 当创建匿名共享内存时，所有默认的内存都是 pinned 状态的，只有用户告诉 Ashmem 驱动程序要 unpin 某一块内存时，Ashmem 驱动程序才会把这块内存 unpin。
☑ 用户告知 Ashmem 驱动程序重新 pin 某一块前面被 unpin 过的内存块，这样能够将此内存从 unpinned 状态转换为 pinned 状态。

函数 ashmem_pin_unpin()在文件 kernel/goldfish/ashmem.c 中实现，具体的实现代码如下所示。

```
static int ashmem_pin_unpin(struct ashmem_area *asma, unsigned long cmd,
			    void __user *p)
{
	struct ashmem_pin pin;
	size_t pgstart, pgend;
	int ret = -EINVAL;

	if (unlikely(!asma->file))
		return -EINVAL;

	if (unlikely(copy_from_user(&pin, p, sizeof(pin))))
		return -EFAULT;
```

```
	/* per custom, you can pass zero for len to mean "everything onward" */
	if (!pin.len)
		pin.len = PAGE_ALIGN(asma->size) - pin.offset;

	if (unlikely((pin.offset | pin.len) & ~PAGE_MASK))
		return -EINVAL;

	if (unlikely(((__u32) -1) - pin.offset < pin.len))
		return -EINVAL;

	if (unlikely(PAGE_ALIGN(asma->size) < pin.offset + pin.len))
		return -EINVAL;

	pgstart = pin.offset / PAGE_SIZE;
	pgend = pgstart + (pin.len / PAGE_SIZE) - 1;

	mutex_lock(&ashmem_mutex);

	switch (cmd) {
	case ASHMEM_PIN:
		ret = ashmem_pin(asma, pgstart, pgend);
		break;
	case ASHMEM_UNPIN:
		ret = ashmem_unpin(asma, pgstart, pgend);
		break;
	case ASHMEM_GET_PIN_STATUS:
		ret = ashmem_get_pin_status(asma, pgstart, pgend);
		break;
	}

	mutex_unlock(&ashmem_mutex);

	return ret;
}
```

由此可见，执行 ASHMEM_PIN 操作的目标对象必须是一块处于 unpinned 状态的内存块。

函数 ashmem_unpin()的功能是解锁某一块匿名共享内存，具体处理流程如下所示。

- ☑ 在遍历 asma->unpinned_list 列表时，查找当前处于 unpinned 状态的内存块是否与将要 unpin 的内存块[pgstart, pgend]相交，如果相交，则通过执行合并操作调整 pgstart 和 pgend 的大小。
- ☑ 调用函数 range_del()删除原来已经被 unpinned 过的内存块。
- ☑ 调用函数 range_alloc()重新 unpinned 调整过后的内存块[pgstart, pgend]，此时新的内存块[pgstart, pgend]已经包含了刚才所有被删掉的 unpinned 状态的内存。
- ☑ 如果找到相交的内存块并且调整了 pgstart 和 pgend 的大小，需要重新扫描 asma->unpinned_list 列表。原因是新的内存块[pgstart, pgend]可能与前后的处于 unpinned 状态的内存块发生相交。

函数 ashmem_unpin()在文件 kernel/goldfish/ashmem.c 中定义，具体的实现代码如下所示。

```
static int ashmem_unpin(struct ashmem_area *asma, size_t pgstart, size_t pgend)
{
```

```
	struct ashmem_range *range, *next;
	unsigned int purged = ASHMEM_NOT_PURGED;

restart:
	list_for_each_entry_safe(range, next, &asma->unpinned_list, unpinned) {
		/* short circuit: this is our insertion point */
		if (range_before_page(range, pgstart))
			break;

		/*
		 * The user can ask us to unpin pages that are already entirely
		 * or partially pinned. We handle those two cases here
		 */
		if (page_range_subsumed_by_range(range, pgstart, pgend))
			return 0;
		if (page_range_in_range(range, pgstart, pgend)) {
			pgstart = min_t(size_t, range->pgstart, pgstart),
			pgend = max_t(size_t, range->pgend, pgend);
			purged |= range->purged;
			range_del(range);
			goto restart;
		}
	}
	return range_alloc(asma, range, purged, pgstart, pgend);
}
```

range_before_page()的操作是一个宏定义，功能是判断 range 描述的内存块是否在 page 页面之前，如果是则表示结束整个描述。asma->unpinned_list 列表是按照页面号从大到小进行排列的，并且每一块被 unpin 的内存都是不相交的。range_before_page()的定义代码如下所示。

```
#define range_before_page(range, page) \
  ((range)->pgend < (page))
```

page_range_subsumed_by_range()的操作也是一个宏定义，功能是判断内存块是否包含了[start, end]这个内存块，如果包含，则说明当前要 unpin 的内存块已经处于 unpinned 状态。如果什么也不用操作，则直接返回。page_range_subsumed_by_range()的定义代码如下所示。

```
#define page_range_subsumed_by_range(range, start, end) \
  (((range)->pgstart <= (start)) && ((range)->pgend >= (end)))
```

page_range_in_range()的操作也是一个宏定义，其功能是判断内存块 [start,end]是否互相包含或者相交。page_range_in_range()的定义代码如下所示。

```
#define page_range_in_range(range, start, end) \
  (page_in_range(range, start) || page_in_range(range, end) || \
   page_range_subsumes_range(range, start, end))
```

page_range_subsumed_by_range()的操作也是一个宏定义，功能是判断内存块 range 是否包含内存块[start, end]。page_range_subsumed_by_range()的定义代码如下所示。

```
#define page_range_subsumed_by_range(range, start, end) \
  (((range)->pgstart <= (start)) && ((range)->pgend >= (end)))
```

page_in_range()的操作也是一个宏定义，功能是判断内存块地址 page 是否包含在内存块 range 中。page_in_range()的定义代码如下所示。

```
#define page_in_range(range, page) \
 (((range)->pgstart <= (page)) && ((range)->pgend >= (page)))
```

再看函数 range_del()，其功能是从 asma->unpinned_list 中删除内存块，并判断它是否在 lru 列表中。函数 range_del()的具体实现代码如下所示。

```
static void range_del(struct ashmem_range *range)
{
        list_del(&range->unpinned);
        if (range_on_lru(range))
                lru_del(range);
        kmem_cache_free(ashmem_range_cachep, range);
}
```

再看函数 lru_del()，内存块的状态 purged 值为 ASHMEM_NOT_PURGED，表示现在没有收回对应的物理页面，那么内存块就位于 lru 列表中，则使用函数 lru_del()删除这个内存块。函数 lru_del()的具体实现代码如下所示。

```
static inline void lru_del(struct ashmem_range *range)
{
        list_del(&range->lru);
        lru_count -= range_size(range);
}
```

再看在函数 ashmem_unpin()中调用的 range_alloc()函数，其功能是从 slab 缓冲区 ashmem_range_cachep 中分配一个 ashmem_range，并进行相应的初始化处理。然后放在对应的列表 ashmem_area->unpinned_list 中，并判断这个 range 的 purged 是否处于 ASHMEM_NOT_PURGED 状态，如果是则要把它放在 lru 列表中。函数 range_alloc()在文件 kernel/goldfish/ashmem.c 中实现，具体的实现代码如下所示。

```
static int range_alloc(struct ashmem_area *asma,
                       struct ashmem_range *prev_range, unsigned int purged,
                       size_t start, size_t end)
{
       struct ashmem_range *range;
       range = kmem_cache_zalloc(ashmem_range_cachep, GFP_KERNEL);
       if (unlikely(!range))
              return -ENOMEM;
       range->asma = asma;
       range->pgstart = start;
       range->pgend = end;
       range->purged = purged;
       list_add_tail(&range->unpinned, &prev_range->unpinned);
       if (range_on_lru(range))
              lru_add(range);
```

```
    return 0;
}
```

再看函数 lru_add()，其功能是将未被回收的已解锁内存块添加到全局列表 ashmem_lru_list 中。函数 lru_add()在文件 kernel/goldfish/ashmem.c 中实现，具体的实现代码如下所示。

```
static inline void lru_add(struct ashmem_range *range)
{
	list_add_tail(&range->lru, &ashmem_lru_list);
	lru_count += range_size(range);
}
```

再看函数 ashmem_pin()，其功能是锁定一块匿名共享内存区域。被 pin 的内存块肯定被保存在 unpinned_list 列表中，如果不在则什么都不用做。要想判断在 unpinned_list 列表中是否存在 pin 的内存块，需要通过遍历 asma->unpinned_list 列表的方式找出与之相交的内存块。函数 ashmem_pin()在文件 kernel/goldfish/ashmem.c 中实现，具体的实现代码如下所示。

```
static int ashmem_pin(struct ashmem_area *asma, size_t pgstart, size_t pgend)
{
	struct ashmem_range *range, *next;
	int ret = ASHMEM_NOT_PURGED;
	list_for_each_entry_safe(range, next, &asma->unpinned_list, unpinned) {
		if (range_before_page(range, pgstart))
			break;
		if (page_range_in_range(range, pgstart, pgend)) {
			ret |= range->purged;
			if (page_range_subsumes_range(range, pgstart, pgend)) {
				range_del(range);
				continue;
			}
			if (range->pgstart >= pgstart) {
				range_shrink(range, pgend + 1, range->pgend);
				continue;
			}
			if (range->pgend <= pgend) {
				range_shrink(range, range->pgstart, pgstart-1);
				continue;
			}
			range_alloc(asma, range, range->purged,
					pgend + 1, range->pgend);
			range_shrink(range, range->pgstart, pgstart - 1);
			break;
		}
	}
	return ret;
}
```

在上述代码中对重新锁定内存块操作实现了判断，通过 if 语句处理了如下 4 种情形。

☑ 指定要锁定的内存块[start,end]包含了解锁状态的内存块 range，此时只要将解锁状态的内存块 range 从其宿主匿名共享内存的解锁内存块列表 unpinned_list 中删除即可。

- ☑ 合并要锁定内存块[pgstart,pgend]的后半部分和解锁状态内存块 range 的前半部分，此时将解锁状态内存块 range 的开始地址设置为要锁定内存块的末尾地址的下一个页面地址。
- ☑ 合并要锁定内存块[pgstart,pgend]的前半部分和解锁状态内存块 range 的后半部分，此时将解锁状态内存块 range 的末尾地址设置为要锁定内存块的开始地址的下一个页面地址。
- ☑ 设置要锁定内存块[pgstart,pgend]包含在解锁状态内存块 range 中。

再看函数 range_shrink()，其功能是设置 range 描述的内存块的起始页面号，如果还存在于 lru 列表中，则需要调整在 lru 列表中的总页面数大小。函数 range_shrink()在文件 kernel/goldfish/ashmem.c 中实现，具体的实现代码如下所示。

```
static inline void range_shrink(struct ashmem_range *range,
                    size_t start, size_t end)
{
    size_t pre = range_size(range);

    range->pgstart = start;
    range->pgend = end;

    if (range_on_lru(range))
        lru_count -= pre - range_size(range);
}
```

3.2.8 回收内存块

接下来开始看最后一步：回收匿名共享内存块。先回到前面介绍的初始化步骤，分析 Ashmem 驱动初始化函数 ashmem_init()，此函数会调用函数 register_shrinker()向内存管理系统注册一个内存回收算法函数，具体的实现代码如下所示。

```
static struct shrinker ashmem_shrinker = {
    .shrink = ashmem_shrink,
    .seeks = DEFAULT_SEEKS * 4,
};
static int __init ashmem_init(void)
{
    ...
    register_shrinker(&ashmem_shrinker);
    printk(KERN_INFO "ashmem: initialized\n");
    return 0;
}
```

其实在 Linux 内核程序中，当系统内存不够用时，内存管理系统就会通过调用内存回收算法的方式将最近没有用过的内存删除，将这些内存从物理内存中清除，这样可以增加物理内存的容量。所以在 Android 系统中也借用了这种机制，当内存管理系统回收内存时会调用函数 ashmem_shrink()以执行内存回收操作。函数 ashmem_shrink()在文件 kernel/goldfish/ashmem.c 中实现，具体的实现代码如下所示。

```
static int ashmem_shrink(struct shrinker *s, struct shrink_control *sc)
{
    struct ashmem_range *range, *next;
```

```
    /* We might recurse into filesystem code, so bail out if necessary */
    if (sc->nr_to_scan && !(sc->gfp_mask & __GFP_FS))
        return -1;
    if (!sc->nr_to_scan)
        return lru_count;
    mutex_lock(&ashmem_mutex);
    list_for_each_entry_safe(range, next, &ashmem_lru_list, lru) {
        loff_t start = range->pgstart * PAGE_SIZE;
        loff_t end = (range->pgend + 1) * PAGE_SIZE;
        do_fallocate(range->asma->file,
                FALLOC_FL_PUNCH_HOLE | FALLOC_FL_KEEP_SIZE,
                start, end - start);
        range->purged = ASHMEM_WAS_PURGED;
        lru_del(range);
        sc->nr_to_scan -= range_size(range);
        if (sc->nr_to_scan <= 0)
            break;
    }
    mutex_unlock(&ashmem_mutex);
    return lru_count;
}
```

3.3　分析 C++访问接口层

如果想在 Android 进程之间共享一个完整的匿名共享内存块，可以通过调用接口 MemoryHeapBase 的方式来实现。如果只是想在进程之间共享匿名共享内存块中的一部分，可以通过调用接口 MemoryBase 来实现。本节将详细分析 C++访问接口层的基本源码。

3.3.1　接口 MemoryHeapBase

接口 MemoryBase 以接口 MemoryHeapBase 为基础，这两个接口都可以作为一个 Binder 对象在进程之间进行传输。因为接口 MemoryHeapBase 是一个 Binder 对象，所以拥有 Server 端对象（必须实现一个 BnInterface 接口）和 Client 端引用（必须要实现一个 BpInterface 接口）的概念。

1．服务器端实现

接口 MemoryHeapBase 在 Server 端的实现过程中，可以将所有涉及的类分为如下 3 种类型。

☑ 业务相关类：即和匿名共享内存操作相关的类，包括 MemoryHeapBase、BnMemoryHeap 和 IMemoryHeap。

☑ Binder 进程通信类：即和 Binder 进程通信机制相关的类，包括 IInterface、BnInterface、IBinder、BBinder、ProcessState 和 IPCThreadState。

☑ 智能指针类：RefBase。

在上述 3 种类型中，Binder 进程通信类和智能指针类将在本书后面的章节中进行讲解。在接口 IMemoryBase 中定义了操作匿名共享内存的几个方法，此接口在文件 frameworks/native/include/binder/IMemory.h 中定义，定义代码如下所示。

```
class IMemoryHeap : public IInterface
{
public:
    DECLARE_META_INTERFACE(MemoryHeap);

    enum {
        READ_ONLY    = 0x00000001
    };

    virtual int getHeapID() const = 0;
    virtual void* getBase() const = 0;
    virtual size_t getSize() const = 0;
    virtual uint32_t getFlags() const = 0;
    virtual uint32_t getOffset() const = 0;

    int32_t heapID() const { return getHeapID(); }
    void*    base() const  { return getBase(); }
    size_t  virtualSize() const { return getSize(); }
};
```

在上述定义代码中有如下 3 个重要的成员函数。

☑ getHeapID()：功能是获得匿名共享内存块的打开文件描述符。

☑ getBase()：功能是获得匿名共享内存块的基地址，通过这个地址可以在程序中直接访问这块共享内存。

☑ getSize()：功能是获得匿名共享内存块的大小。

类 BnMemoryHeap 是一个本地对象类，当 Client 端引用请求 Server 端对象执行命令时，Binder 系统就会调用类 BnMemoryHeap 的成员函数 onTransact()执行具体的命令。函数 onTransact()在文件 frameworks/native/libs/binder/IMemory.cpp 中定义，具体实现代码如下所示。

```
status_t BnMemory::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    switch(code) {
        case GET_MEMORY: {
            CHECK_INTERFACE(IMemory, data, reply);
            ssize_t offset;
            size_t size;
            reply->writeStrongBinder( getMemory(&offset, &size)->asBinder() );
            reply->writeInt32(offset);
            reply->writeInt32(size);
            return NO_ERROR;
        } break;
        default:
            return BBinder::onTransact(code, data, reply, flags);
    }
}
```

类 MemoryHeapBase 继承了类 BnMemoryHeap，作为 Binder 机制中的 Server 角色需要实现 IMemoryBase 接口，主要功能是实现类 IMemoryBase 中列出的成员函数，描述了一块匿名共享内存服

务。类在文件 frameworks/native/include/binder/MemoryHeapBase.h 中定义，具体实现代码如下所示。

```
class MemoryHeapBase : public virtual BnMemoryHeap
{
public:
    enum {
        READ_ONLY = IMemoryHeap::READ_ONLY,
        DONT_MAP_LOCALLY = 0x00000100,
        NO_CACHING = 0x00000200
    };
    MemoryHeapBase(int fd, size_t size, uint32_t flags = 0, uint32_t offset = 0);
    MemoryHeapBase(const char* device, size_t size = 0, uint32_t flags = 0);
    MemoryHeapBase(size_t size, uint32_t flags = 0, char const* name = NULL);

    virtual ~MemoryHeapBase();
    virtual int getHeapID() const;
    virtual void* getBase() const;

    virtual size_t getSize() const;
    virtual uint32_t getFlags() const;
    virtual uint32_t getOffset() const;

    const char* getDevice() const;
    void dispose();
    status_t setDevice(const char* device) {
        if (mDevice == 0)
            mDevice = device;
        return mDevice ? NO_ERROR : ALREADY_EXISTS;
    }

protected:
            MemoryHeapBase();
    status_t init(int fd, void *base, int size,
            int flags = 0, const char* device = NULL);

private:
    status_t mapfd(int fd, size_t size, uint32_t offset = 0);

    int mFD;    //是一个文件描述符，是在打开设备文件/dev/ashmem 后得到的，能够描述一个匿名共享内存块
    size_t mSize;           //内存块的大小
    void* mBase;            //内存块的映射地址
    uint32_t mFlags;        //内存块的访问保护位
    const char* mDevice;
    bool mNeedUnmap;
    uint32_t mOffset;
};
```

类 MemoryHeapBase 在文件 frameworks/native/libs/binder/MemoryHeapBase.cpp 中实现，其核心功能是包含了一块匿名共享内存。对应代码如下所示。

```
MemoryHeapBase::MemoryHeapBase(size_t size, uint32_t flags, char const * name)
    : mFD(-1), mSize(0), mBase(MAP_FAILED), mFlags(flags),
      mDevice(0), mNeedUnmap(false), mOffset(0)
{
    const size_t pagesize = getpagesize();
    size = ((size + pagesize-1) & ~(pagesize-1));
    int fd = ashmem_create_region(name == NULL ? "MemoryHeapBase" : name, size);
    ALOGE_IF(fd<0, "error creating ashmem region: %s", strerror(errno));
    if (fd >= 0) {
        if (mapfd(fd, size) == NO_ERROR) {
            if (flags & READ_ONLY) {
                ashmem_set_prot_region(fd, PROT_READ);
            }
        }
    }
}
```

各个参数的具体说明如下所示。

- ☑ size：表示要创建的匿名共享内存的大小。
- ☑ flags：设置这块匿名共享内存的属性，例如可读写、只读等。
- ☑ name：此参数只是作为调试信息使用的，用于标识匿名共享内存的名字，可以是空值。

接下来看 MemoryHeapBase 的成员函数 mapfd()，其功能是将得到的匿名共享内存的文件描述符映射到进程地址空间。函数 mapfd()在文件 frameworks/native/libs/binder/MemoryHeapBase.cpp 中定义，具体实现代码如下所示。

```
status_t MemoryHeapBase::mapfd(int fd, size_t size, uint32_t offset)
{
    if (size == 0) {
#ifdef HAVE_ANDROID_OS
        pmem_region reg;
        int err = ioctl(fd, PMEM_GET_TOTAL_SIZE, &reg);
        if (err == 0)
            size = reg.len;
#endif
        if (size == 0) { // try fstat
            struct stat sb;
            if (fstat(fd, &sb) == 0)
                size = sb.st_size;
        }
    }

    if ((mFlags & DONT_MAP_LOCALLY) == 0) {//条件为 true 时执行系统调用 mmap()来执行内存映射的操作
        void* base = (uint8_t*)mmap(
0,//表示由内核来决定这个匿名共享内存文件在进程地址空间的起始位置
 size,//表示要映射的匿名共享内存文件的大小
PROT_READ|PROT_WRITE,//表示这个匿名共享内存是可读写的
 MAP_SHARED,
 fd, //指定要映射的匿名共享内存的文件描述符
 offset//表示要从这个文件的哪个偏移位置开始映射
```

```
);
        if (base == MAP_FAILED) {
            ALOGE("mmap(fd=%d, size=%u) failed (%s)",
                    fd, uint32_t(size), strerror(errno));
            close(fd);
            return -errno;
        }
        //ALOGD("mmap(fd=%d, base=%p, size=%lu)", fd, base, size);
        mBase = base;
        mNeedUnmap = true;
    } else  {
        mBase = 0;
        mNeedUnmap = false;
    }
    mFD = fd;
    mSize = size;
    mOffset = offset;
    return NO_ERROR;
}
```

这样在调用函数 mapfd()后，会进入到内核空间的 Ashmem 驱动程序模块中执行函数 ashmem_map()。有关函数 ashmem_map()的具体实现过程，在 3.2 节的内容中进行了详细讲解。

最后看成员函数 getHeapID()、getBase()和 getSize()的具体实现，其实现代码如下所示。

```
int MemoryHeapBase::getHeapID() const {
    return mFD;
}
void* MemoryHeapBase::getBase() const {
    return mBase;
}
size_t MemoryHeapBase::getSize() const {
    return mSize;
}
```

2．客户端实现

接口 MemoryHeapBase 在客户端的实现过程中，可以将所有涉及的类分为如下 3 种类型。

- ☑ 业务相关类：即和匿名共享内存操作相关的类，包括 BpMemoryHeap 和 IMemoryHeap。
- ☑ Binder 进程通信类：即和 Binder 进程通信机制相关的类，包括 IInterface、BpInterface、IBinder、BpBinder、ProcessState、BpRefBase 和 IPCThreadState。
- ☑ 智能指针类：RefBase。

在上述 3 种类型中，Binder 进程通信类和智能指针类将在本书后面的章节中进行讲解，在本章将重点介绍业务相关类。

类 BpMemoryHeap 是类 MemoryHeapBase 在 Client 端进程的远程接口类，当 Client 端进程从 Service Manager 获得了一个 MemoryHeapBase 对象的引用后，会在本地创建一个 BpMemoryHeap 对象来表示这个引用。类 BpMemoryHeap 是从 RefBase 类继承的，也要实现 IMemoryHeap 接口，可以和智能指针结合使用。

类 BpMemoryHeap 在文件 frameworks/native/libs/binder/IMemory.cpp 中定义，具体实现代码如下所示。

```
class BpMemoryHeap : public BpInterface<IMemoryHeap>
{
public:
    BpMemoryHeap(const sp<IBinder>& impl);
    virtual ~BpMemoryHeap();

    virtual int getHeapID() const;
    virtual void* getBase() const;
    virtual size_t getSize() const;
    virtual uint32_t getFlags() const;
    virtual uint32_t getOffset() const;

private:
    friend class IMemory;
    friend class HeapCache;

    static inline sp<IMemoryHeap> find_heap(const sp<IBinder>& binder) {
        return gHeapCache->find_heap(binder);
    }
    static inline void free_heap(const sp<IBinder>& binder) {
        gHeapCache->free_heap(binder);
    }
    static inline sp<IMemoryHeap> get_heap(const sp<IBinder>& binder) {
        return gHeapCache->get_heap(binder);
    }
    static inline void dump_heaps() {
        gHeapCache->dump_heaps();
    }

    void assertMapped() const;
    void assertReallyMapped() const;

    mutable volatile int32_t mHeapId;
    mutable void* mBase;
    mutable size_t mSize;
    mutable uint32_t mFlags;
    mutable uint32_t mOffset;
    mutable bool mRealHeap;
    mutable Mutex mLock;
};
```

类 BpMemoryHeap 对应的构造函数是 BpMemoryHeap()，具体实现代码如下所示。

```
BpMemoryHeap::BpMemoryHeap(const sp<IBinder>& impl)
    : BpInterface<IMemoryHeap>(impl),
        mHeapId(-1), mBase(MAP_FAILED), mSize(0), mFlags(0), mRealHeap(false)
{
}
```

在使用成员函数 getHeapID()、getBase()和 getSize()之前，需要通过调用函数 assertMapped()来确保

在 Client 端已经准备好了匿名共享内存。成员函数 getHeapID()、getBase()和 getSize()的具体实现代码如下所示。

```
int BpMemoryHeap::getHeapID() const {
    assertMapped();
    return mHeapId;
}

void* BpMemoryHeap::getBase() const {
    assertMapped();
    return mBase;
}

size_t BpMemoryHeap::getSize() const {
    assertMapped();
    return mSize;
}
```

函数 assertMapped()在文件 frameworks/native/libs/binder/IMemory.cpp 中定义，具体实现代码如下所示。

```
void BpMemoryHeap::assertMapped() const
{
    if (mHeapId == -1) {
        sp<IBinder> binder(const_cast<BpMemoryHeap*>(this)->asBinder());
        sp<BpMemoryHeap> heap(static_cast<BpMemoryHeap*>(find_heap(binder).get()));
        heap->assertReallyMapped();
        if (heap->mBase != MAP_FAILED) {
            Mutex::Autolock _l(mLock);
            if (mHeapId == -1) {
                mBase   = heap->mBase;
                mSize   = heap->mSize;
                android_atomic_write( dup( heap->mHeapId ), &mHeapId );
            }
        } else {
            free_heap(binder);
        }
    }
}
```

类 HeapCache 在文件 frameworks/native/libs/binder/IMemory.cpp 中定义，具体实现代码如下所示。

```
class HeapCache : public IBinder::DeathRecipient
{
public:
    HeapCache();
    virtual ~HeapCache();

    virtual void binderDied(const wp<IBinder>& who);
```

```
	sp<IMemoryHeap> find_heap(const sp<IBinder>& binder);
	void free_heap(const sp<IBinder>& binder);
	sp<IMemoryHeap> get_heap(const sp<IBinder>& binder);
	void dump_heaps();

private:
	struct heap_info_t {
		sp<IMemoryHeap> heap;
		int32_t count;
	};

	void free_heap(const wp<IBinder>& binder);

	Mutex mHeapCacheLock;
	KeyedVector< wp<IBinder>, heap_info_t > mHeapCache;
};
```

在上述代码中定义了成员变量 mHeapCache，功能是维护进程内的所有 BpMemoryHeap 对象。另外还提供了函数 find_heap()和 get_heap()来查找内部所维护的 BpMemoryHeap 对象，这两个函数的具体说明如下所示。

- ☑ 函数 find_heap()：如果在 mHeapCache 中找不到相应的 BpMemoryHeap 对象，则把 BpMemoryHeap 对象加入到 mHeapCache 中。
- ☑ 函数 get_heap()：不会自动把 BpMemoryHeap 对象加入到 mHeapCache 中。

接下来看函数 find_heap()，首先以传进来的参数 binder 作为关键字在 mHeapCache 中查找，查找是否存在对应的 heap_info 对象 info。

- ☑ 存在 info：增加引用计数 info.count 的值，表示此 BpBinder 对象多了一个使用者。
- ☑ 不存在 info：创建一个放到 mHeapCache 中的 heap_info 对象 info。

函数 find_heap()在文件 frameworks/native/libs/binder/IMemory.cpp 中定义，具体实现代码如下所示。

```
sp<IMemoryHeap> HeapCache::find_heap(const sp<IBinder>& binder)
{
	Mutex::Autolock _l(mHeapCacheLock);
	ssize_t i = mHeapCache.indexOfKey(binder);
	if (i>=0) {
		heap_info_t& info = mHeapCache.editValueAt(i);
		ALOGD_IF(VERBOSE,
				"found binder=%p, heap=%p, size=%d, fd=%d, count=%d",
				binder.get(), info.heap.get(),
				static_cast<BpMemoryHeap*>(info.heap.get())->mSize,
				static_cast<BpMemoryHeap*>(info.heap.get())->mHeapId,
				info.count);
		android_atomic_inc(&info.count);
		return info.heap;
	} else {
		heap_info_t info;
		info.heap = interface_cast<IMemoryHeap>(binder);
		info.count = 1;
```

```
            //ALOGD("adding binder=%p, heap=%p, count=%d",
            //binder.get(), info.heap.get(), info.count);
            mHeapCache.add(binder, info);
            return info.heap;
        }
}
```

由上述实现代码可知，函数 find_heap()是 BpMemoryHeap 的成员函数，能够调用全局变量 gHeapCache 执行查找的操作。对应的实现代码如下所示。

```
class BpMemoryHeap : public BpInterface<IMemoryHeap>
{
...
private:
    static inline sp<IMemoryHeap> find_heap(const sp<IBinder>& binder) {
        return gHeapCache->find_heap(binder);
    }
```

通过调用函数 find_heap()得到 BpMemoryHeap 对象中的函数 assertReallyMapped()，这样可以确认其内部的匿名共享内存是否已经映射到进程空间。函数 assertReallyMapped()在文件 frameworks/native/libs/binder/IMemory.cpp 中定义，具体实现代码如下所示。

```
void BpMemoryHeap::assertReallyMapped() const
{
    if (mHeapId == -1) {
        Parcel data, reply;
        data.writeInterfaceToken(IMemoryHeap::getInterfaceDescriptor());
        status_t err = remote()->transact(HEAP_ID, data, &reply);
        int parcel_fd = reply.readFileDescriptor();
        ssize_t size = reply.readInt32();
        uint32_t flags = reply.readInt32();
        uint32_t offset = reply.readInt32();

        ALOGE_IF(err, "binder=%p transaction failed fd=%d, size=%ld, err=%d (%s)",
                asBinder().get(), parcel_fd, size, err, strerror(-err));

        int fd = dup( parcel_fd );
        ALOGE_IF(fd==-1, "cannot dup fd=%d, size=%ld, err=%d (%s)",
                parcel_fd, size, err, strerror(errno));

        int access = PROT_READ;
        if (!(flags & READ_ONLY)) {
            access |= PROT_WRITE;
        }

        Mutex::Autolock _l(mLock);
        if (mHeapId == -1) {
            mRealHeap = true;
            mBase = mmap(0, size, access, MAP_SHARED, fd, offset);
```

```
                if (mBase == MAP_FAILED) {
                    ALOGE("cannot map BpMemoryHeap (binder=%p), size=%ld, fd=%d (%s)",
                            asBinder().get(), size, fd, strerror(errno));
                    close(fd);
                } else {
                    mSize = size;
                    mFlags = flags;
                    mOffset = offset;
                    android_atomic_write(fd, &mHeapId);
                }
            }
        }
}
```

3.3.2 接口 MemoryBase

接口 MemoryBase 是建立在接口 MemoryHeapBase 的基础上的，两者都可以作为一个 Binder 对象在进程之间实现数据共享。

1. 在 Server 端的实现

首先分析类 MemoryBase 在 Server 端的实现，MemoryBase 在 Server 端只是简单地封装了 MemoryHeapBase 的实现。类 MemoryBase 在 Server 端的实现和类 MemoryHeapBase 在 Server 端的实现类似，只需在整个类图结构中实现如下转换即可。

☑ 把类 IMemory 转换成类 IMemoryHeap。

☑ 把类 BnMemory 转换成类 BnMemoryHeap。

☑ 把类 MemoryBase 转换成类 MemoryHeapBase。

类 IMemory 在文件 frameworks/native/include/binder/IMemory.h 中实现，功能是定义类 MemoryBase 所需要的实现接口。类 IMemory 的实现代码如下所示。

```
class IMemory : public IInterface
{
public:
    DECLARE_META_INTERFACE(Memory);
    virtual sp<IMemoryHeap> getMemory(ssize_t* offset=0, size_t* size=0) const = 0;
    void* fastPointer(const sp<IBinder>& heap, ssize_t offset) const;
    void* pointer() const;
    size_t size() const;
    ssize_t offset() const;
};
```

在类 IMemory 中定义了如下成员函数。

☑ getMemory()：功能是获取内部的 MemoryHeapBase 对象的 IMemoryHeap 接口。

☑ pointer()：功能是获取内部所维护的匿名共享内存的基地址。

☑ size()：功能是获取内部所维护的匿名共享内存的大小。

☑ offset()：功能是获取内部所维护的匿名共享内存在整个匿名共享内存中的偏移量。

类 IMemory 在本身定义过程中实现了 3 个成员函数：pointer()、size()和 offset()，其子类 MemoryBase 只需实现成员函数 getMemory()即可。类 IMemory 的具体实现在文件 frameworks/native/libs/binder/IMemory.cpp 中定义，具体实现代码如下所示。

```
void* IMemory::pointer() const {
    ssize_t offset;
    sp<IMemoryHeap> heap = getMemory(&offset);
    void* const base = heap!=0 ? heap->base() : MAP_FAILED;
    if (base == MAP_FAILED)
        return 0;
    return static_cast<char*>(base) + offset;
}

size_t IMemory::size() const {
    size_t size;
    getMemory(NULL, &size);
    return size;
}
ssize_t IMemory::offset() const {
    ssize_t offset;
    getMemory(&offset);
    return offset;
}
```

类 MemoryBase 是一个本地 Binder 对象类，在文件 frameworks/native/include/binder/MemoryBase.h 中声明，具体定义代码如下所示。

```
class MemoryBase : public BnMemory
{
public:
    MemoryBase(const sp<IMemoryHeap>& heap, ssize_t offset, size_t size);
    virtual ~MemoryBase();
    virtual sp<IMemoryHeap> getMemory(ssize_t* offset, size_t* size) const;
protected:
    size_t getSize() const { return mSize; }
    ssize_t getOffset() const { return mOffset; }
    const sp<IMemoryHeap>& getHeap() const { return mHeap; }
private:
    size_t mSize;
    ssize_t mOffset;
    sp<IMemoryHeap> mHeap;
};
};
#endif
```

类 MemoryBase 的具体实现在文件 frameworks/native/libs/binder/MemoryBase.cpp 中定义，具体实现代码如下所示。

```
MemoryBase::MemoryBase(const
 sp<IMemoryHeap>& heap,//指向 MemoryHeapBase 对象，真正的匿名共享内存就是由它来维护的
```

```
  ssize_t offset,//表示这个 MemoryBase 对象所要维护的这部分匿名共享内存在整个匿名共享内存块中的起始位置
  size_t size//表示这个 MemoryBase 对象所要维护的这部分匿名共享内存的大小
)
    : mSize(size), mOffset(offset), mHeap(heap)
{
}
//功能是返回内部的 MemoryHeapBase 对象的 IMemoryHeap 接口
//如果传进来的参数 offset 和 size 不为 NULL
//会把其内部维护的匿名共享内存在整个匿名共享内存块中的偏移位置以及这部分匿名共享内存的大小返回给调用者
sp<IMemoryHeap> MemoryBase::getMemory(ssize_t* offset, size_t* size) const
{
    if (offset) *offset = mOffset;
    if (size)   *size = mSize;
    return mHeap;
}
```

2．MemoryBase 类在 Client 端的实现

再来看 MemoryBase 类在 Client 端的实现。类 MemoryBase 在 Client 端的实现与类 MemoryHeapBase 在 Client 端的实现类似，只需要进行如下类转换即可成为 MemoryHeapBase 在 Client 端的实现。

☑ 把类 IMemory 转换成类 IMemoryHeap。

☑ 把类 BpMemory 转换成类 BpMemoryHeap。

类 BpMemory 用于描述类 MemoryBase 服务的代理对象，在文件 frameworks/native/libs/binder/IMemory.cpp 中定义，具体实现代码如下所示。

```
class BpMemory : public BpInterface<IMemory>
{
public:
    BpMemory(const sp<IBinder>& impl);
    virtual ~BpMemory();
    virtual sp<IMemoryHeap> getMemory(ssize_t* offset=0, size_t* size=0) const;

private:
    mutable sp<IMemoryHeap> mHeap;  //类型为 IMemoryHeap，指向的是一个 BpMemoryHeap 对象
    mutable ssize_t mOffset;//表示 BpMemory 对象所要维护的匿名共享内存在整个匿名共享内存块中的起始位置
    mutable size_t mSize;          //表示这个 BpMemory 对象所要维护的这部分匿名共享内存的大小
};
```

类 BpMemory 中的成员函数 getMemory()在文件 frameworks/native/libs/binder/IMemory.cpp 中定义，具体实现代码如下所示。

```
sp<IMemoryHeap> BpMemory::getMemory(ssize_t* offset, size_t* size) const
{
    if (mHeap == 0) {
        Parcel data, reply;
        data.writeInterfaceToken(IMemory::getInterfaceDescriptor());
        if (remote()->transact(GET_MEMORY, data, &reply) == NO_ERROR) {
            sp<IBinder> heap = reply.readStrongBinder();
```

```
            ssize_t o = reply.readInt32();
            size_t s = reply.readInt32();
            if (heap != 0) {
                mHeap = interface_cast<IMemoryHeap>(heap);
                if (mHeap != 0) {
                    mOffset = o;
                    mSize = s;
                }
            }
        }
    }
    if (offset) *offset = mOffset;
    if (size) *size = mSize;
    return mHeap;
}
```

如果成员变量 mHeap 的值为 NULL，表示此 BpMemory 对象还没有建立好匿名共享内存，此时会调用一个 Binder 进程去 Server 端请求匿名共享内存信息。通过引用信息中的 Server 端的 MemoryHeapBase 对象的引用 heap，可以在 Client 端进程中创建一个 BpMemoryHeap 远程接口，最后将这个 BpMemoryHeap 远程接口保存在成员变量 mHeap 中，同时从 Server 端获得的信息还包括这块匿名共享内存在整个匿名共享内存中的偏移位置以及大小。

3.4 分析 Java 访问接口层

分析完匿名共享内存的 C++访问接口层后，本节开始分析其 Java 访问接口层的实现过程。在 Android 5.0 应用程序框架层中，通过接口 MemoryFile 来实现封装匿名共享内存文件的创建和使用功能。在 Android 5.0 中，接口 MemoryFile 在文件 frameworks/base/core/java/android/os/MemoryFile.java 中定义，具体实现代码如下所示。

```
public class MemoryFile
{
    private static String TAG = "MemoryFile";
    private static final int PROT_READ = 0x1;
    private static final int PROT_WRITE = 0x2;

    private static native FileDescriptor native_open(String name, int length) throws IOException;
    private static native int native_mmap(FileDescriptor fd, int length, int mode)
            throws IOException;
    private static native void native_munmap(int addr, int length) throws IOException;
    private static native void native_close(FileDescriptor fd);
    private static native int native_read(FileDescriptor fd, int address, byte[] buffer,
            int srcOffset, int destOffset, int count, boolean isUnpinned) throws IOException;
    private static native void native_write(FileDescriptor fd, int address, byte[] buffer,
            int srcOffset, int destOffset, int count, boolean isUnpinned) throws IOException;
    private static native void native_pin(FileDescriptor fd, boolean pin) throws IOException;
```

```
	private static native int native_get_size(FileDescriptor fd) throws IOException;

	private FileDescriptor mFD;
	private int mAddress;
	private int mLength;
	private boolean mAllowPurging = false;

	public MemoryFile(String name, int length) throws IOException {
		mLength = length;
		mFD = native_open(name, length);
		if (length > 0) {
			mAddress = native_mmap(mFD, length, PROT_READ | PROT_WRITE);
		} else {
			mAddress = 0;
		}
	}
```

在上述代码中，构造方法 MemoryFile()以指定的字符串调用了 JNI 方法 native_open()，目的是建立一个匿名共享内存文件，这样可以得到一个文件描述符。然后使用这个文件描述符为参数调用 JNI 方法 natvie_mmap()，并把匿名共享内存文件映射到进程空间中，这样就可以通过映射得到地址空间的方式直接访问内存数据。

JNI 函数 android_os_MemoryFile_get_size()在文件 frameworks/base/core/jni/android_os_MemoryFile.cpp 中定义，具体实现代码如下所示。

```
static jint android_os_MemoryFile_get_size(JNIEnv* env, jobject clazz,
		jobject fileDescriptor) {
	int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
	int result = ashmem_get_size_region(fd);
	if (result < 0) {
		if (errno == ENOTTY) {
			return (jint) -1;
		}
		jniThrowIOException(env, errno);
		return (jint) -1;
	}
	return (jint) result;
}
```

JNI 函数 android_os_MemoryFile_open()在文件 frameworks/base/core/jni/android_os_MemoryFile.cpp 中定义，具体实现代码如下所示。

```
static jobject android_os_MemoryFile_open(JNIEnv* env, jobject clazz, jstring name, jint length)
{
	const char* namestr = (name ? env->GetStringUTFChars(name, NULL) : NULL);

	int result = ashmem_create_region(namestr, length);

	if (name)
```

```
        env->ReleaseStringUTFChars(name, namestr);

    if (result < 0) {
        jniThrowException(env, "java/io/IOException", "ashmem_create_region failed");
        return NULL;
    }

    return jniCreateFileDescriptor(env, result);
}
```

JNI 函数 android_os_MemoryFile_mmap()在文件 frameworks/base/core/jni/android_os_MemoryFile.cpp 中定义，具体实现代码如下所示。

```
static jint android_os_MemoryFile_mmap(JNIEnv* env, jobject clazz, jobject fileDescriptor,
        jint length, jint prot)
{
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    jint result = (jint)mmap(NULL, length, prot, MAP_SHARED, fd, 0);
    if (!result)
        jniThrowException(env, "java/io/IOException", "mmap failed");
    return result;
}
```

在文件 frameworks/base/core/java/android/os/MemoryFile.java 中，类 MemoryFile 的成员函数 readBytes()功能是读取某一块匿名共享内存的内容。具体实现代码如下所示。

```
public int readBytes(byte[] buffer, int srcOffset, int destOffset, int count)
        throws IOException {
    if (isDeactivated()) {
        throw new IOException("Can't read from deactivated memory file.");
    }
    if (destOffset < 0 || destOffset > buffer.length || count < 0
            || count > buffer.length - destOffset
            || srcOffset < 0 || srcOffset > mLength
            || count > mLength - srcOffset) {
        throw new IndexOutOfBoundsException();
    }
     return native_read(mFD, mAddress, buffer, srcOffset, destOffset, count, mAllowPurging);
}
```

在文件 frameworks/base/core/java/android/os/MemoryFile.java 中，类 MemoryFile 的成员函数 writeBytes()功能是写入某一块匿名共享内存的内容。具体实现代码如下所示。

```
public void writeBytes(byte[] buffer, int srcOffset, int destOffset, int count)
            throws IOException {
        if (isDeactivated()) {
            throw new IOException("Can't write to deactivated memory file.");
        }
        if (srcOffset < 0 || srcOffset > buffer.length || count < 0
                || count > buffer.length - srcOffset
```

```
                || destOffset < 0 || destOffset > mLength
                || count > mLength - destOffset) {
            throw new IndexOutOfBoundsException();
        }
        native_write(mFD, mAddress, buffer, srcOffset, destOffset, count, mAllowPurging);
}
```

在文件 frameworks/base/core/java/android/os/MemoryFile.java 中，类 MemoryFile 的成员函数 isDeactivated()功能是保证匿名共享内存已经被映射到进程的地址空间中。具体实现代码如下所示。

```
void deactivate() {
        if (!isDeactivated()) {
            try {
                native_munmap(mAddress, mLength);
                mAddress = 0;
            } catch (IOException ex) {
                Log.e(TAG, ex.toString());
            }
}
}
private boolean isDeactivated() {
        return mAddress == 0;
}
```

JNI 函数 native_read()和 native_write()分别由位于 C++层的函数 android_os_MemoryFile_read()和 android_os_MemoryFile_write()实现，这两个 C++的函数在文件 frameworks/base/core/jni/android_os_MemoryFile.cpp 中定义，具体实现代码如下所示。

```
static jint android_os_MemoryFile_read(JNIEnv* env, jobject clazz,
        jobject fileDescriptor, jint address, jbyteArray buffer, jint srcOffset, jint destOffset,
        jint count, jboolean unpinned)
{
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    if (unpinned && ashmem_pin_region(fd, 0, 0) == ASHMEM_WAS_PURGED) {
        ashmem_unpin_region(fd, 0, 0);
        jniThrowException(env, "java/io/IOException", "ashmem region was purged");
        return -1;
    }
    env->SetByteArrayRegion(buffer, destOffset, count, (const jbyte *)address + srcOffset);
    if (unpinned) {
        ashmem_unpin_region(fd, 0, 0);
    }
    return count;
}
static jint android_os_MemoryFile_write(JNIEnv* env, jobject clazz,
        jobject fileDescriptor, jint address, jbyteArray buffer, jint srcOffset, jint destOffset,
        jint count, jboolean unpinned)
{
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
```

```
    if (unpinned && ashmem_pin_region(fd, 0, 0) == ASHMEM_WAS_PURGED) {
        ashmem_unpin_region(fd, 0, 0);
        jniThrowException(env, "java/io/IOException", "ashmem region was purged");
        return -1;
    }
    env->GetByteArrayRegion(buffer, srcOffset, count, (jbyte *)address + destOffset);
    if (unpinned) {
        ashmem_unpin_region(fd, 0, 0);
    }
    return count;
}
```

第 4 章　硬件抽象层架构详解

在 Android 系统中，硬件抽象层（Hardware Abstract Layer，HAL）在用户空间中运行。HAL 能够向下屏蔽硬件驱动模块的实现细节，向上提供硬件访问服务。通过硬件抽象层，Android 系统通过如下两层来支持硬件设备。

☑　第一层在用户空间中实现。

☑　第二层在内核空间中实现。

本章将详细讲解 Android 5.0 中 HAL 源码的基本知识，为读者学习本书后面更高深的知识打下基础。

4.1　HAL 基础

HAL 层（硬件抽象层）是位于操作系统内核与硬件电路之间的接口层，其目的在于将硬件抽象化。该层隐藏了特定平台的硬件接口细节，为操作系统提供虚拟硬件平台，使其具有硬件无关性，这样就可以在多种平台上进行移植。从软硬件测试的角度来看，软硬件的测试工作都可分别基于硬件抽象层来完成，从此使软硬件测试工作的并行进行成为可能。本节将简要介绍 HAL 的基础知识。

4.1.1　推出 HAL 的背景

在 Android 系统中，推出 HAL 的目的是为了保护一些硬件提供商的知识产权，为了避开 Linux 的 GPL 束缚。Google 架构师的思路是把控制硬件的动作都放到了 Android HAL 中，而 Linux Driver（驱动）仅仅负责完成一些简单的数据交互作用，甚至把硬件寄存器空间直接映射到 User Space（用户空间）。而 Android 系统是基于 Aparch 的 License，因此硬件厂商可以只提供二进制代码，所以说 Android 只是一个开放的平台，并不是一个开源的平台。也许正是因为 Android 不遵从 GPL，所以 Greg Kroah-Hartman 才在 2.6.33 内核将 Andorid 驱动从 Linux 中删除，当然从后来 Linux 3.3 版本开始又将 Android 重新纳入。GPL 和硬件厂商目前还是有着无法弥合的裂痕。Android 想要把这个问题处理好也是不容易的。

Android 系统为什么要把对硬件的支持划分为两层来实现呢？具体来说有如下两个原因。

（1）Linux 内核源代码是遵循 GPL1 协议的，即如果在 Android 系统所使用的 Linux 内核中添加或者修改了代码，那么就必须将它们公开。因此，如果 Android 系统像其他的 Linux 系统一样，把对硬件的支持完全实现在硬件驱动模块中，那么就必须将这些硬件驱动模块源代码公开，这样就可能损害移动设备厂商的利益，因为这相当于暴露了硬件的实现细节和参数。

（2）Android 系统源代码是遵循 Apache License 2 协议的，允许移动设备厂商添加或者修改 Android 系统源代码，而又不必公开这些代码。因此，如果把对硬件的支持完全实现在 Android 系统的用户空间中，那么就可以隐藏硬件的实现细节和参数。然而，这是无法做到的，因为只有内核空间才有特权操作硬件设备。一个折中的解决方案便是将对硬件的支持分别实现在内核空间和用户空间中，其中，

内核空间仍然是以硬件驱动模块的形式来支持，不过它只提供简单的硬件访问通道；而用户空间以硬件抽象层模块的形式来支持，封装了硬件的实现细节和参数。这样就可以保护移动设备厂商的利益。

在 Android 系统中可以分为如下 6 种 HAL。

☑ 上层软件。
☑ 内部以太网。
☑ 内部通信 CLIENT。
☑ 用户接入口。
☑ 虚拟驱动，设置管理模块。
☑ 内部通信 Server。

在 Android 系统中，定义硬件抽象层接口的代码具有以下 5 个特点。

☑ 硬件抽象层具有与硬件的密切相关性。
☑ 硬件抽象层具有与操作系统无关性。
☑ 接口定义的功能应包含硬件或系统所需硬件支持的所有功能。
☑ 接口定义简单明了，接口函数太多会增加软件模拟的复杂性。
☑ 具有可测性的接口设计有利于系统的软硬件测试和集成。

在 Android 源码中，HAL 主要被保存在下面的目录中。

☑ libhardware_legacy：过去的目录，采取了链接库模块观念来架构。
☑ libhardware：新版的目录，被调整为用 HAL stub 观念来架构。
☑ ril：是 Radio 接口层。
☑ msm7k：和 QUAL 平台相关的信息。

4.1.2　HAL 的基本结构

在 Android 系统中，HAL 的位置结构如图 4-1 所示。



图 4-1　HAL 层结构

从图 4-1 所示的结构图可以看出，HAL 的功能是把 Android Framework（Android 框架）与 Linux 内核（Android 内核）隔离。这样 Android 可以不过度依赖 Linux Kernel，从而以在不考虑驱动程序的前提下进行 Framework 层的应用开发工作。在 HAL 层主要包含了 GPS、Vibrator、Wi-Fi、Copybit、Audio、Camera、Lights、Ril 和 Overlay 等模块。

目前 Android 的 HAL 层仍然分布在不同的位置，所以诸如 Camera、Wi-Fi 等目录并不包含所有的

HAL 程序代码。HAL 架构成熟前的结构如图 4-2 所示，现在 HAL 层的结构如图 4-3 所示。

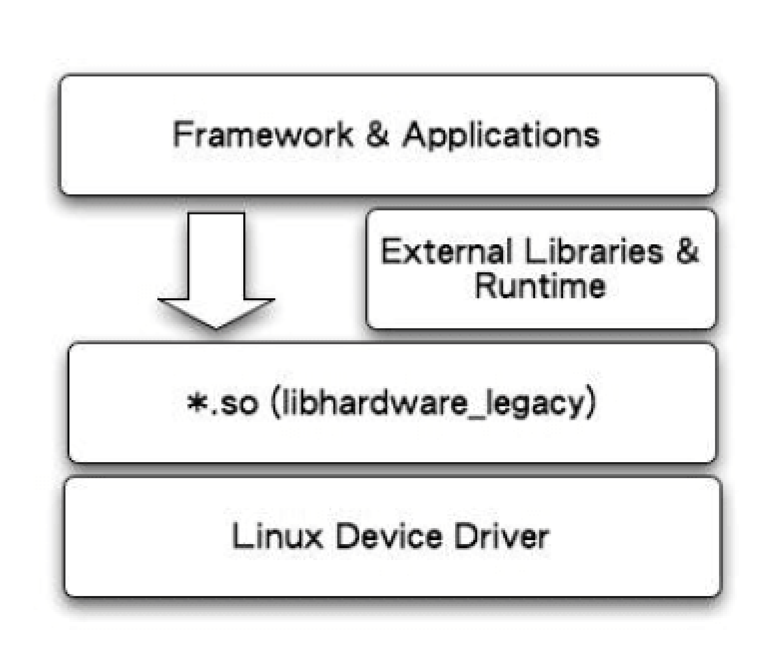


图 4-2　成熟前的 HAL 架构

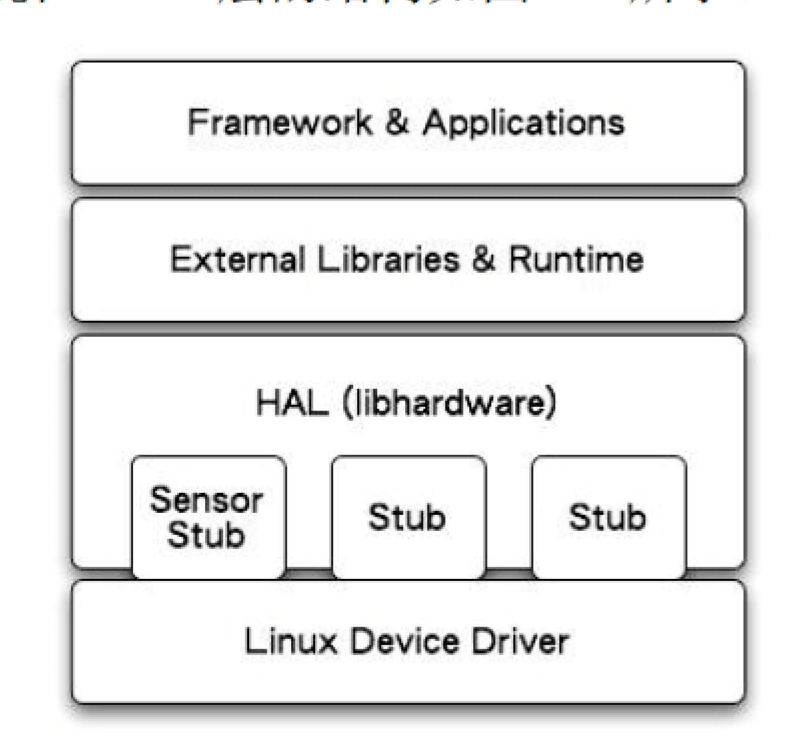


图 4-3　现在的 HAL 架构

从现在 HAL 层的结构可以看出，当前的 HAL stub 模式是一种代理人（proxy）的概念，虽然 stub 仍以*.so 档的形式存在，但是 HAL 已经将*.so 档隐藏了。stub 向 HAL 提供了功能强大的操作函数（Operations），而 runtime 则从 HAL 获取特定模块（stub）的函数，然后再回调这些操作函数。这种以 Indirect Function Call 模式的架构，让 HAL stub 变成了一种“包含”关系，也就是说在 HAL 中包含了许多 stub（代理人）。Runtime 只要说明 module ID（类型）就可以取得操作函数。在当前的 HAL 模式中，Android 定义了 HAL 层结构框架，这样通过接口访问硬件时就形成了统一的调用方式。

注意：HAL_legacy 和 HAL 的对比

为了使读者明白过去结构和现在结构的差别，接下来将对 HAL_legacy 和 HAL 做一个对比。

（1）HAL_legacy

这是过去 HAL 的模块，采用共享库形式，在编译时会调用到。由于采用 function call 形式来调用，因此可被多个进程使用，但会被 mapping 到多个进程空间中造成浪费，同时需要考虑代码能否安全重入的问题（thread safe）。

（2）HAL

这是新式的 HAL，采用了 HAL module 和 HAL stub 结合形式。HAL stub 不是一个共享库，在编译时上层只拥有访问 HAL stub 的函数指针，并不需要 HAL stub。在上层通过 HAL module 提供的统一接口获取并操作 HAL stub，所以文件只会被映射到一个进程，而不会存在重复映射和重入问题。

在 Android 系统中，HAL 层的源码结构如下所示。

（1）/hardware/libhardware_legacy/：旧的 HAL 架构、采取链接库模块的方式。

（2）/hardware/libhardware：新的 HAL 架构，调整为 HAL stub，具体目录的结构如下所示。

☑ /hardware/libhardware/hardware.c：编译成 libhardware.s，置于/system/lib。

☑ /hardware/libhardware/include/hardware 目录下包含如下头文件。

➢ hardware.h：通用硬件模块头文件。

➢ copybit.h：copybit 模块头文件。

➢ gralloc.h：gralloc 模块头文件。

➢ lights.h：背光模块头文件。

> ➢ overlay.h：overlay 模块头文件。
> ➢ qemud.h：qemud 模块头文件。
> ➢ sensors.h：传感器模块头文件。

☑ /hardware/libhardware/modules：在此目录下定义了很多硬件模块，例如/hardware/msm7k、/hardware/qcom、/hardware/ti、/device/Samsung 和/device/moto，这些是各个厂商平台相关的 HAL。

4.2　分析 HAL module 架构

Android 5.0 的 HAL 采用 HAL module 和 HAL stub 结合的形式进行架构，HAL stub 不是一个 Share Library（共享程序），在编译时上层只拥有访问 HAL stub 的函数指针，并不需要 HAL stub。上层通过 HAL module 提供的统一接口获取并操作 HAL stub，so 文件只会被 mapping 到一个进程，也不存在重复 mapping 和重入问题。

在 Android 5.0 系统中，HAL module 架构主要分为如下 3 个结构体。

☑ struct hw_module_t
☑ struct hw_module_methods_t
☑ struct hw_device_t

上述 3 个结构的继承关系如图 4-4 所示。

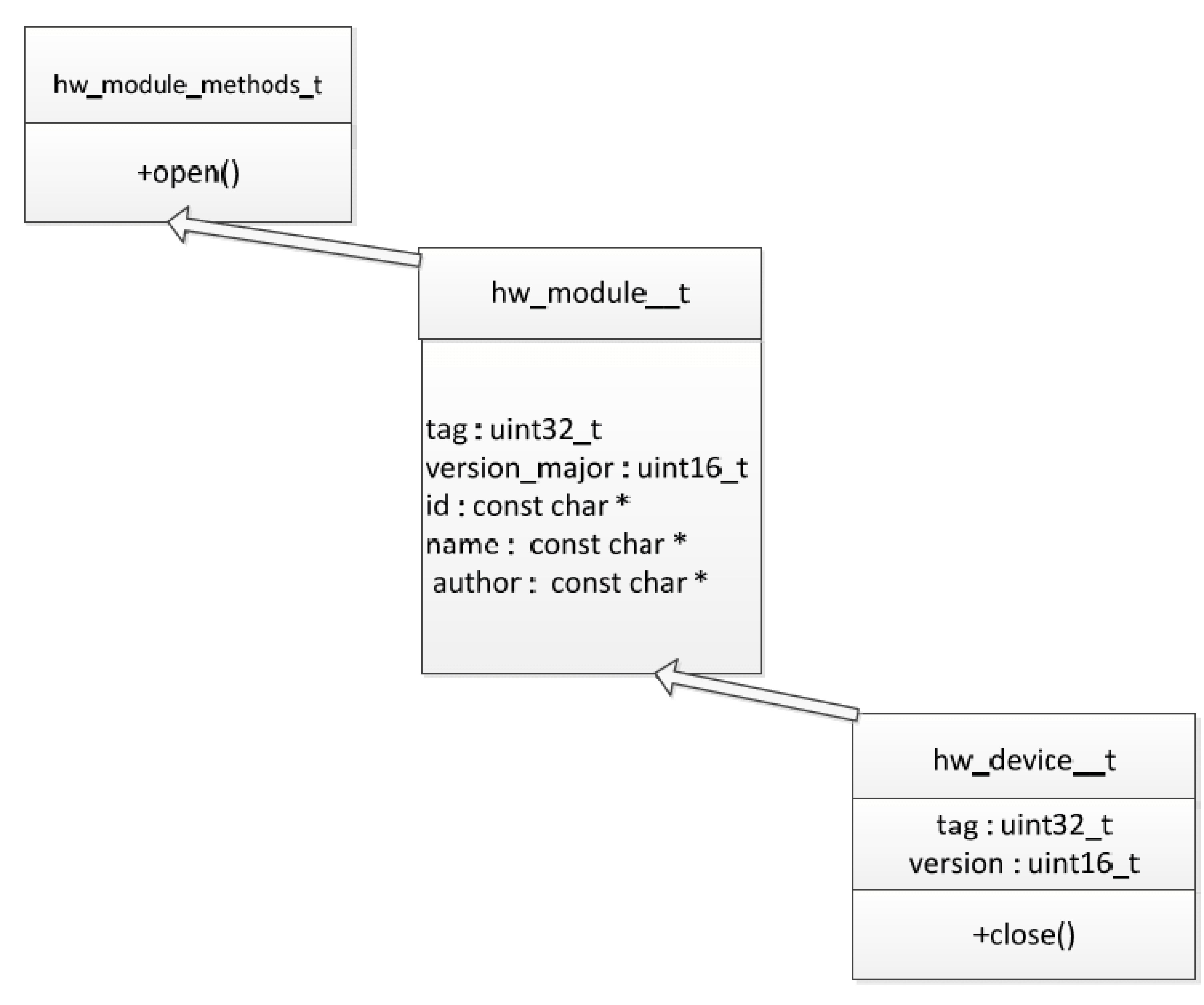


图 4-4　Android HAL 结构的继承关系

以上 3 个抽象概念在文件 hardware.c 中进行了具体描述，而 HAL 模块的源代码保存在 harware 目录中。对于不同的 hardware 的 HAL，对应的 lib 命名规则是 id.variant.so，例如 gralloc.msm7k.so 表示其 id 是 gralloc，msm7k 是 variant。variant 的取值范围是在该文件中定义的 variant_keys 对应的值。

4.2.1 hw_module_t

结构 hw_module_t 在文件 hardware/libhardware/include/hardware/hardware.h 中定义，具体实现代码如下所示。

```
typedef struct hw_module_t {
    uint32_t tag;
    uint16_t module_api_version;
#define version_major module_api_version
    uint16_t hal_api_version;
#define version_minor hal_api_version
    const char *id;
    const char *name;
    const char *author;
    struct hw_module_methods_t* methods;
    void* dso;
    uint32_t reserved[32-7];

} hw_module_t;
```

在结构体 hw_module_t 中，读者需要注意如下 5 点。

（1）在结构体 hw_module_t 的定义前面有一段注释，意思是，硬件抽象层中的每一个模块都必须自定义一个硬件抽象层模块结构体，而且它的第一个成员变量的类型必须为 hw_module_t。

（2）硬件抽象层中的每一个模块都必须存在一个导出符号 HAL_MODULE_IFNO_SYM，即 HMI，它指向一个自定义的硬件抽象层模块结构体。后面在分析硬件抽象层模块的加载过程时，将会看到这个导出符号的意义。

（3）结构体 hw_module_t 的成员变量 tag 的值必须设置为 HARDWARE_MODULE_TAG，即设置为一个常量值（'H' <<24 | 'W'<<16 | 'M'<<8 | 'T'），用来标志这是一个硬件抽象层模块结构体。

（4）结构体 hw_module_t 的成员变量 dso 用来保存加载硬件抽象层模块后得到的句柄值。前面提到，每一个硬件抽象层模块都对应有一个动态链接库文件。加载硬件抽象层模块的过程实际上就是调用 dlopen()函数来加载与其对应的动态链接库文件的过程。在调用 dlclose()函数来卸载这个硬件抽象层模块时，要用到这个句柄值，因此，在加载时需要将其保存起来。

（5）结构体 hw_module_t 的成员变量 methods 定义了一个硬件抽象层模块的操作方法列表，其类型为 hw_module_methods_t，接下来就介绍它的定义。hw_module_methods_t 的定义代码如下所示。

```
typedef struct hw_module_methods_t {
    /** Open a specific device */
    int (*open)(const struct hw_module_t* module, const char* id,
            struct hw_device_t** device);
} hw_module_methods_t;
```

4.2.2 hw_module_methods_t

结构 hw_module_methods_t 在文件 hardware/libhardware/include/hardware/hardware.h 中定义，具体

实现代码如下所示。

```
typedef struct hw_module_methods_t {
    /** Open a specific device */
    int (*open)(const struct hw_module_t* module, const char* id,
            struct hw_device_t** device);

} hw_module_methods_t;
```

在结构体 hw_module_methods_t 中只有一个成员变量，它是一个函数指针，用来打开硬件抽象层模块中的硬件设备。其中，参数 module 表示要打开的硬件设备所在的模块；参数 id 表示要打开的硬件设备的 ID；参数 device 是一个输出参数，用来描述一个已经打开的硬件设备。由于一个硬件抽象层模块可能会包含多个硬件设备，因此在调用结构体 hw_module_methods_t 的成员变量 open，打开一个硬件设备时需要指定其 ID。

4.2.3　hw_device_t

结构 hw_device_t 在文件 hardware/libhardware/include/hardware/hardware.h 中定义，具体实现代码如下所示。

```
typedef struct hw_device_t {
    uint32_t tag;
    uint32_t version;
    struct hw_module_t* module;
    uint32_t reserved[12];
    int (*close)(struct hw_device_t* device);

} hw_device_t;
```

在 Android 系统中，硬件抽象层中的硬件设备使用结构体 hw_device_t 来描述。定义 hw_device_t 的代码如下所示。

```
typedef struct hw_device_t {
    uint32_t tag;
    uint32_t version;
    struct hw_module_t* module;
    uint32_t reserved[12];
    int (*close)(struct hw_device_t* device);
} hw_device_t;
```

在结构体 hw_device_t 中，需要注意如下 3 点。

（1）硬件抽象层模块中的每一个硬件设备都必须自定义一个硬件设备结构体，而且它的第一个成员变量的类型必须为 hw_device_t。

（2）结构体 hw_device_t 的成员变量 tag 的值必须设置为 HARDWARE_DEVICE_TAG，即设置为一个常量值（'H' <<24 | 'W'<<16 | 'D'<<8 | 'T'），用来标志这是一个硬件抽象层中的硬件设备结构体。

（3）结构体 hw_device_t 的成员变量 close 是一个函数指针，用来关闭一个硬件设备。

4.3 分析文件 hardware.c

文件 hardware.c 是文件 hardware.h 的具体实现。本节将详细分析 Android 5.0 HAL 模块中文件 hardware.c 的基本源码。

4.3.1 寻找动态链接库的地址

函数 hw_get_module()能够根据模块 ID 寻找硬件模块动态链接库的地址，然后调用函数 load()打开动态链接库，并从中获取硬件模块结构体地址。执行后首先得到是根据固定的符号 HAL_MODULE_INFO_SYM 寻找到结构体 hw_module_t，然后是用 hw_moule_t 中 hw_module_methods_t 结构体成员函数提供的结构 open 打开相应的模块，并同时进行初始化操作。因为用户在调用 open()时通常都会传入一个指向 hw_device_t 指针的指针。这样函数 open()将对模块的操作函数结构保存到结构体 hw_device_t 中，用户通过它可以和模块进行交互。

函数 hw_get_module()的实现代码如下所示。

```
int hw_get_module(const char *id, const struct hw_module_t **module)
    {
        int status;
        int i;
        const struct hw_module_t *hmi = NULL;
        char prop[PATH_MAX];
        char path[PATH_MAX];
    /* Loop through the configuration variants looking for a module */
…
        for (i=0 ; i<HAL_VARIANT_KEYS_COUNT+1 ; i++) {
…
```

4.3.2 数组 variant_keys

在函数 hw_get_module()中需要用到数组 variant_keys，因为 HAL_VARIANT_KEYS_COUNT 表示数组 variant_keys 的大小。定义数组 variant_keys 的代码如下所示。

```
static const char *variant_keys[] = {
    "ro.hardware",    /* This goes first so that it can pick up a different file on the emulator */
    "ro.product.board",
    "ro.board.platform",
    "ro.arch"
};
```

然后通过此数组，并使用如下代码得到操作权限。

```
if (i < HAL_VARIANT_KEYS_COUNT) {
    if (property_get(variant_keys[i], prop, NULL) == 0) {
        continue;
    }
```

此处的 variant_keys[i]对应有 3 个值，分别是 trout、msm7k 和 ARMv6。

接下来通过如下代码将路径和文件名保存到 path。

```
snprintf(path, sizeof(path), "%s/%s.%s.so",
     HAL_LIBRARY_PATH, id, prop);
```

通过上述代码，把 HAL_LIBRARY_PATH/id.***.so 保存到 path 中，其中"***"就是上面 variant_keys 中各个元素所对应的值。

4.3.3　载入相应的库

载入相应的库，并把它们的 HMI 保存到 module 中。具体代码如下所示。

```
          } else {
               snprintf(path, sizeof(path), "%s/%s.default.so",
                    HAL_LIBRARY_PATH, id);
          }
          if (access(path, R_OK)) {
               continue;
          }
          /* we found a library matching this id/variant */
          break;
     }

     status = -ENOENT;
     if (i < HAL_VARIANT_KEYS_COUNT+1) {
          /* load the module, if this fails, we're doomed, and we should not try
          * to load a different variant. */
          status = load(id, path, module);//load 相应库，并把它们的 HMI 保存到 module 中
     }

    return status;
}
```

4.3.4　获得 hw_module_t 结构体

通过函数 load()打开相应的库并获得 hw_module_t 结构体，具体实现代码如下所示。

```
static int load(const char *id,
          const char *path,
          const struct hw_module_t **pHmi)
{
     int status;
     void *handle;
     struct hw_module_t *hmi;
handle = dlopen(path, RTLD_NOW);//打开相应的库
     if (handle == NULL) {
               char const *err_str = dlerror();
```

```
                LOGE("load: module=%s\n%s", path, err_str?err_str:"unknown");
                status = -EINVAL;
                goto done;
        }

        const char *sym = HAL_MODULE_INFO_SYM_AS_STR;
        hmi = (struct hw_module_t *)dlsym(handle, sym);//获得 hw_module_t 结构体
        if (hmi == NULL) {
            LOGE("load: couldn't find symbol %s", sym);
            status = -EINVAL;
            goto done;
        }

        /* Check that the id matches */
        if (strcmp(id, hmi->id) != 0) {//只是一个 check
            LOGE("load: id=%s != hmi->id=%s", id, hmi->id);
            status = -EINVAL;
            goto done;
        }

        hmi->dso = handle;

        /* success */
        status = 0;
done:
        if (status != 0) {
            hmi = NULL;
            if (handle != NULL) {
                dlclose(handle);
                handle = NULL;
            }
        } else {
            LOGV("loaded HAL id=%s path=%s hmi=%p handle=%p",
                    id, path, *pHmi, handle);
        }

        *pHmi = hmi;//得到 hw_module_t

        return status;
}
```

4.4 分析硬件抽象层的加载过程

每一个硬件抽象层模块在内核中都对应一个驱动程序，硬件抽象层模块就是通过这些驱动程序来访问硬件设备的，它们是通过读写设备文件来进行通信的。硬件抽象层中的模块接口源文件一般保存在 hardware/libhardware 目录中，其目录结构如图 4-5 所示。

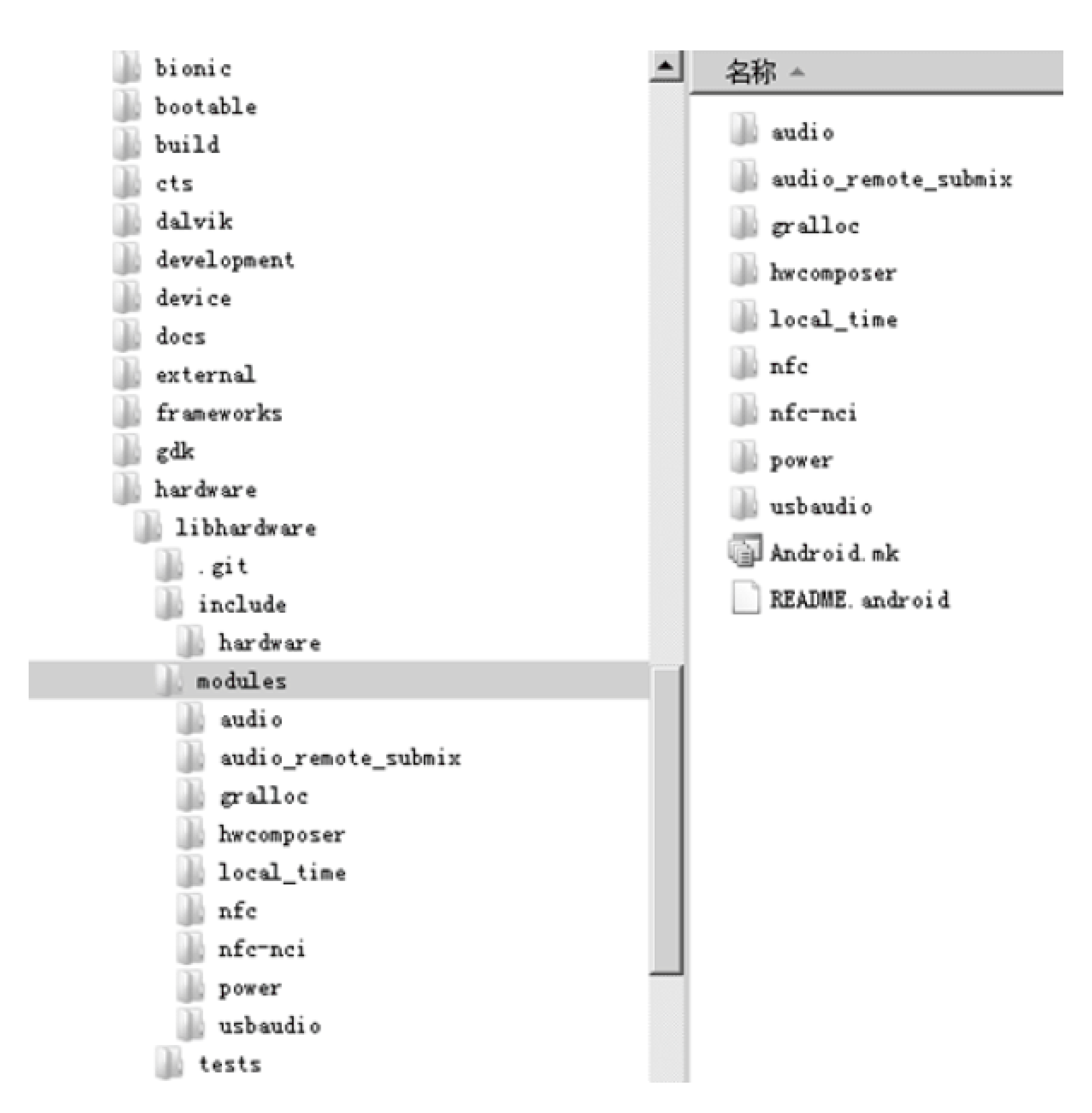


图 4-5　libhardware 目录

Android 系统中的硬件抽象层模块是由系统统一加载的，当调用者需要加载这些模块时，只要指定它们的 ID 值就可以了。在 Android 硬件抽象层中，负责加载硬件抽象层模块的函数是 hw_get_module()，此函数在文件 hardware/libhardware/include/hardware/hardware.h 中定义。

函数 hw_get_module()有 id 和 module 两个参数。其中，id 是输入参数，表示要加载的硬件抽象层模块 ID；module 是输出参数，如果加载成功，那么它指向一个自定义的硬件抽象层模块结构体。函数的返回值是一个整数，如果等于 0，则表示加载成功；如果小于 0，则表示加载失败。函数 hw_get_module()的具体实现代码如下所示。

```
int hw_get_module(const char *id, const struct hw_module_t **module)
{
    return hw_get_module_by_class(id, NULL, module);
}
```

函数 hw_get_module()在文件 hardware/libhardware/hardware.c 中实现，其中数组 variant_keys 用来组装要加载的硬件抽象层模块的文件名称。常量 HAL_VARIANT_KEYS_COUNT 表示数组 variant_keys 的大小。宏 HAL_LIBRARY_PATH1 和 HAL_LIBRARY_PATH2 用来定义要加载的硬件抽象层模块文件所在的目录。第 32～50 行的 for 循环根据数组 variant_keys 在 HAL_LIBRARY_PATH1 和 HAL_LIBRARY_PATH2 目录中检查对应的硬件抽象层模块文件是否存在，如果存在则结束 for 循环；第 56 行调用 load()函数来执行加载硬件抽象层模块的操作。函数 hw_get_module()的具体实现代码如下所示。

```
16  int hw_get_module(const char *id, const struct hw_module_t **module)
17  {
18  int status;
19  int i;
20  const struct hw_module_t *hmi = NULL;
21  char prop[PATH_MAX];
```

```
22 char path[PATH_MAX];
23
24 /*
25 * Here we rely on the fact that calling dlopen multiple times on
26 * the same .so will simply increment a refcount (and not load
27 * a new copy of the library).
28 * We also assume that dlopen() is thread-safe.
29 */
30
31 /* Loop through the configuration variants looking for a module */
32 for (i=0 ; i<HAL_VARIANT_KEYS_COUNT+1 ; i++) {
33 if (i < HAL_VARIANT_KEYS_COUNT) {
34 if (property_get(variant_keys[i], prop, NULL) == 0) {
35 continue;
36 }
37
38 snprintf(path, sizeof(path), "%s/%s.%s.so",
39 HAL_LIBRARY_PATH1, id, prop);
40 if (access(path, R_OK) == 0) break;
41
42 snprintf(path, sizeof(path), "%s/%s.%s.so",
43 HAL_LIBRARY_PATH2, id, prop);
44 if (access(path, R_OK) == 0) break;
45 } else {
46 snprintf(path, sizeof(path), "%s/%s.default.so",
47 HAL_LIBRARY_PATH1, id);
48 if (access(path, R_OK) == 0) break;
49 }
50 }
51
52 status = -ENOENT;
53 if (i < HAL_VARIANT_KEYS_COUNT+1) {
54 /* load the module, if this fails, we're doomed, and we should not try
55 * to load a different variant. */
56 status = load(id, path, module);
57 }
58
59 return status;
60 }
```

编译好的模块文件位于 out/target/product/generic/system/lib/hw 目录中，而这个目录经过打包后，就对应于设备上的/system/lib/hw 目录。宏 HAL_LIBRARY_PATH2 所定义的目录为/vendor/lib/hw，用来保存设备厂商所提供的硬件抽象层模块接口文件。

在上述第 56 行代码中，调用函数 load()执行硬件抽象层模块的加载操作，此函数的具体实现代码如下所示。

```
static int load(const char *id,
const char *path,
const struct hw_module_t **pHmi)
{
```

```
int status;
void *handle;
struct hw_module_t *hmi;

/*
* load the symbols resolving undefined symbols before
* dlopen returns. Since RTLD_GLOBAL is not or'd in with
* RTLD_NOW the external symbols will not be global
*/
handle = dlopen(path, RTLD_NOW);
if (handle == NULL) {
char const *err_str = dlerror();
LOGE("load: module=%s\n%s", path, err_str?err_str:"unknown");
status = -EINVAL;
goto done;
}

/* Get the address of the struct hal_module_info. */
const char *sym = HAL_MODULE_INFO_SYM_AS_STR;
hmi = (struct hw_module_t *)dlsym(handle, sym);
if (hmi == NULL) {
LOGE("load: couldn't find symbol %s", sym);
status = -EINVAL;
goto done;
}

/* Check that the id matches */
if (strcmp(id, hmi->id) != 0) {
LOGE("load: id=%s != hmi->id=%s", id, hmi->id);
status = -EINVAL;
goto done;
}

hmi->dso = handle;

/* success */
status = 0;

done:
if (status != 0) {
hmi = NULL;
if (handle != NULL) {
dlclose(handle);
handle = NULL;
}
} else {
LOGV("loaded HAL id=%s path=%s hmi=%p handle=%p",
id, path, *pHmi, handle);
}

```

```
55    *pHmi = hmi;
56
57    return status;
58  }
```

在上述代码中，第 14 行调用函数 dlopen()将其加载到内存中。加载完成这个动态链接库文件之后，第 24 行就调用函数 dlsym()来获得里面名称为 HAL_MODULE_INFO_SYM_AS_STR 的符号。这个 HAL_MODULE_INFO_SYM_AS_STR 符号指向的是一个自定义的硬件抽象层模块结构体，包含了对应的硬件抽象层模块的所有信息。HAL_MODULE_INFO_SYM_AS_STR 是一个宏，其值定义为 HMI。根据硬件抽象层模块的编写规范，每一个硬件抽象层模块都必须包含一个名称为 HMI 的符号，而且这个符号的第一个成员变量的类型必须定义为 hw_module_t，因此，第 24 行可以安全地将模块中的 HMI 符号转换为一个 hw_module_t 结构体指针。获得了这个 hw_module_t 结构体指针之后，第 32 行调用 strcmp()函数来验证加载得到的硬件抽象层模块 ID 是否与所要求加载的硬件抽象层模块 ID 一致。如果不一致，则说明出错了，函数返回一个错误值-EINVAL。最后，第 38 行将成功加载后得到的模块句柄值 handle 保存在 hw_module_t 结构体指针 hmi 的成员变量 dso 中，然后将其返回给调用者。

4.5　分析硬件访问服务

当开发好硬件抽象层模块之后，通常还需要在应用程序框架层中实现一个硬件访问服务。硬件访问服务通过硬件抽象层模块来为应用程序提供硬件读写操作。由于硬件抽象层模块是使用 C++语言开发的，而应用程序框架层中的硬件访问服务是使用 Java 语言开发的，因此，硬件访问服务必须通过 Java 本地接口（Java Native Interface，JNI）来调用硬件抽象层模块的接口。本节将详细分析硬件访问服务的基本源码。

4.5.1　定义硬件访问服务接口

Android 系统的硬件访问服务通常运行在系统进程 System5 中，而使用这些硬件访问服务的应用程序运行在另外的进程中，即应用程序需要通过进程间通信机制来访问这些硬件访问服务。Android 系统提供了一种高效的进程间通信机制——Binder 进程间通信机制 6，应用程序就是通过它来访问运行在系统进程 System 中的硬件访问服务的。Binder 进程间通信机制要求提供服务的一方必须实现一个具有跨进程访问能力的服务接口，以便使用服务的一方可以通过这个服务接口来访问它。因此，在实现硬件访问服务之前，首先要定义它的服务接口。

在 Android 5.0 系统中，提供了一种描述语言来定义具有跨进程访问能力的服务接口，这种描述语言称为 Android 接口描述语言（Android Interface Definition Language，AIDL）。以 AIDL 定义的服务接口文件是以 aidl 为后缀名的，在编译时，编译系统会将它们转换成一个 Java 文件，然后再对它们进行编译。本节将使用 AIDL 来定义硬件访问服务接口 IFregService。

在 Android 系统中，通常在 frameworks/base/core/java/android/os 目录中定义硬件访问服务接口，所以把定义了硬件访问服务接口 IFregService 的文件 IFregService.aidl 也保存在这个目录中，其具体代码如下所示。

```
package android.os;
interface IFregService {
void setVal(int val);
int getVal();
}
```

服务接口 IFregService 只定义了两个成员函数，分别是 setVal()和 getVal()。其中，成员函数 setVal()用来往虚拟硬件设备 freg 的寄存器 val 中写入一个整数，而成员函数 getVal()用来从虚拟硬件设备 freg 的寄存器 val 中读出一个整数。

由于服务接口 IFregService 是使用 AIDL 语言描述的，因此需要将其添加到编译脚本文件中，这样编译系统才能将其转换为 Java 文件，然后再对它进行编译。进入到 frameworks/base 目录中，打开里面的 Android.mk 文件，修改 LOCAL_SRC_FILES 变量的值。

```
LOCAL_SRC_FILES += \
...
voip/java/android/net/sip/ISipService.aidl \
core/java/android/os/IFregService.aidl
```

修改这个编译脚本文件之后，就可以使用 mmm 命令对硬件访问服务接口 IFregService 进行编译了。

```
USER@MACHINE:~/Android$ mmm ./frameworks/base/
```

编译后得到的 framework.jar 文件就包含有 IFregService 接口，它继承了 android.os.IInterface 接口。在 IFregService 接口内部，定义了一个 Binder 本地对象类 Stub，它实现了 IFregService 接口，并且继承了 android.os.Binder 类。此外，在 IFregService.Stub 类内部，还定义了一个 Binder 代理对象类 Proxy，它同样也实现了 IFregService 接口。

用 AIDL 定义的服务接口是用来进行进程间通信的，其中，提供服务的进程称为 Server 进程，而使用服务的进程称为 Client 进程。在 Server 进程中，每一个服务都对应有一个 Binder 本地对象，它通过一个桩（Stub）来等待 Client 进程发送进程间通信请求。Client 进程在访问运行 Server 进程中的服务之前，首先要获得它的一个 Binder 代理对象接口（Proxy），然后通过这个 Binder 代理对象接口向它发送进程间通信请求。

4.5.2　具体实现

在 Android 系统中，通常通过 frameworks/base/services/java/com/android/server 目录中的文件实现硬件访问服务。因此把实现了硬件访问服务 FregService 的文件 FregService.java 也保存在这个目录中，其具体内容如下所示。

```
package com.android.server;

import android.content.Context;
import android.os.IFregService;
import android.util.Slog;

public class FregService extends IFregService.Stub {
private static final String TAG = "FregService";
```

```
private int mPtr = 0;

FregService() {
mPtr = init_native();

if(mPtr == 0) {
Slog.e(TAG, "Failed to initialize freg service.");
}
}

public void setVal(int val) {
if(mPtr == 0) {
Slog.e(TAG, "Freg service is not initialized.");
return;
}

setVal_native(mPtr, val);
}

public int getVal() {
if(mPtr == 0) {
Slog.e(TAG, "Freg service is not initialized.");
return 0;
}

return getVal_native(mPtr);
}

private static native int init_native();
private static native void setVal_native(int ptr, int val);
private static native int getVal_native(int ptr);
};
```

在上述代码中，硬件访问服务 FregService 继承了类 IFregService.Stub，并且实现了 IFregService 接口的成员函数 setVal()和 getVal()。其中，成员函数 setVal()通过调用 JNI 方法 setVal_native()来写虚拟硬件设备 freg 的寄存器 val，而成员函数 getVal()调用 JNI 方法 getVal_native()来读虚拟硬件设备 freg 的寄存器 val。在启动硬件访问服务 FregService 时，会通过调用 JNI 函数 init_native()来打开虚拟硬件设备 freg，并且获得它的一个句柄值，保存在成员变量 mPtr 中。如果硬件访问服务 FregService 打开虚拟硬件设备 freg 失败，那么其成员变量 mPtr 的值等于 0；否则，就得到一个大于 0 的句柄值。这个句柄值实际上是指向虚拟硬件设备 freg 在硬件抽象层中的一个设备对象，硬件访问服务 FregService 的成员函数 setVal()和 getVal()在访问虚拟硬件设备 freg 的寄存器 val 时，必须要指定这个句柄值，以便硬件访问服务 FregService 的 JNI 实现可以知道它所要访问的是哪一个硬件设备。

4.6 分析官方实例

谷歌针对 HAL 提供了一个官方实例工程：mokoid，在此工程中提供了一个 LedTest 演示程序，此

程序实例完整演示了 Android 层次结构和 HAL 架构编程的方法和流程。本节将详细分析 mokoid 工程的基本源码。

4.6.1 获取实例工程源码

读者可以从网络中获取 LedTest 示例程序的源码，方法是在 Linux 中使用下面的下载命令。

```
#svn checkout http://mokoid.googlecode.com/svn/trunk/mokoid-read-only
```

下载 mokoid 工程文件后，其目录结构如图 4-6 所示。

图 4-6 mokoid 工程的目录结构

mokiod 工程代码树的具体说明如下所示。

```
.
|-- apps    -- 测试应用程序
|    |-- LedClient -- 直接调用 service 控制硬件
|    |    |-- AndroidManifest.xml
|    |    `-- src
|    |         `-- com
|    |              `-- mokoid
|    |                   `-- LedClient
|    |                        `-- LedClient.java
|    `-- LedTest -- 通过 manager 来控制硬件
|         |-- AndroidManifest.xml
|         `-- src
|              `-- com
|                   `-- mokoid
|                        `-- LedTest
|                             |-- LedSystemServer.java
|                             `-- LedTest.java
|-- frameworks -- 框架代码
|    `-- base
|         |-- core
|         |    `-- java
|         |         `-- mokoid
|         |              `-- hardware
|         |                   |-- ILedService.aidl -- Android Interface Definition Language 代码，提供
LedService 的接口
|         |                   `-- LedManager.java -- LedManager 实现代码
|         `-- service
|              |-- com.mokoid.server.xml
|              |-- java
```

```
|                  |     `-- com
|                  |           `-- mokoid
|                  |                 `-- server
|                  |                       `-- LedService.java -- LedService 的 java 实现代码
|                  `-- jni
|                        `-- com_mokoid_server_LedService.cpp -- LedService 的 jni 实现代码
|-- hardware
    `-- modules
          |-- include
          |     `-- mokoid
          |           `-- led.h
          `-- led
                `-- led.c -- led 实际控制硬件的代码
```

在 Android 系统中需要通过 JNI（Java Native Interface）实现 HAL，因为 JNI 是 Java 程序可以调用 C/C++编写的动态链接库，所以可以使用 C/C++语言编写 HAL，这样做的好处是拥有更高的效率。在 Android 系统中有如下两种访问 HAL 的方式。

（1）Android APP 直接通过 service 调用.so 格式的 JNI，虽然这种方式比较简单高效，但是不正规。

（2）经过 Manager 调用 Service，虽然此方式实现起来比较复杂，但是更符合目前的 Android 框架。在此方法中，在进程 LegManager 和 LedService（Java）中需要通过进程通信的方式实现通信。

在 mokoid 工程中分别实现了上述两种方法，下面将详细介绍这两种方法的具体实现原理。

4.6.2　直接调用 service()方法的实现代码

（1）HAL 层的实现代码

文件 hardware/modules/led/led.c 的实现代码如下所示。

```
#define LOG_TAG "MokoidLedStub"
#include <hardware/hardware.h>
#include <fcntl.h>
#include <errno.h>
#include <cutils/log.h>
#include <cutils/atomic.h>
#include <mokoid/led.h>
/******************************************************************************/
int led_device_close(struct hw_device_t* device)
{
	struct led_control_device_t* ctx = (struct led_control_device_t*)device;
	if (ctx) {
		free(ctx);
	}
	return 0;
}
int led_on(struct led_control_device_t *dev, int32_t led)
```

```
{
        LOGI("LED Stub: set %d on.", led);
        return 0;
}
int led_off(struct led_control_device_t *dev, int32_t led)
{
        LOGI("LED Stub: set %d off.", led);
        return 0;
}
static int led_device_open(const struct hw_module_t* module, const char* name,
              struct hw_device_t** device)
{
        struct led_control_device_t *dev;
        dev = (struct led_control_device_t *)malloc(sizeof(*dev));
        memset(dev, 0, sizeof(*dev));
        dev->common.tag =   HARDWARE_DEVICE_TAG;
        dev->common.version = 0;
        dev->common.module = module;
        dev->common.close = led_device_close;
        dev->set_on = led_on;              //实例化支持的操作
        dev->set_off = led_off;
        *device = &dev->common;        //将实例化的 led_control_device_t 地址返回给 JNI 层
success:
        return 0;
}
static struct hw_module_methods_t led_module_methods = {
       open: led_device_open
};
const struct led_module_t HAL_MODULE_INFO_SYM = {
    //定义此对象相当于向系统注册了一个 ID 为 LED_HARDWARE_MODULE_ID 的 stub
  common: {
              tag: HARDWARE_MODULE_TAG,
              version_major: 1,
              version_minor: 0,
              id: LED_HARDWARE_MODULE_ID,
              name: "Sample LED Stub",
              author: "The Mokoid Open Source Project",
              methods: &led_module_methods,//实现了一个 open 的方法供 JNI 层调用
       }
       /* supporting APIs go here */
};
```

（2）JNI 层的实现代码

文件 frameworks/base/service/jni/com_mokoid_server_LedService.cpp 的实现代码如下所示。

```
struct led_control_device_t *sLedDevice = NULL;
static jboolean mokoid_setOn(JNIEnv* env, jobject thiz, jint led) {
       LOGI("LedService JNI: mokoid_setOn() is invoked.");
       if (sLedDevice == NULL) {
              LOGI("LedService JNI: sLedDevice was not fetched correctly.");
```

```
            return -1;
        } else {
            return sLedDevice->set_on(sLedDevice, led);     //调用 HAL 层的注册方法
        }
}
static jboolean mokoid_setOff(JNIEnv* env, jobject thiz, jint led) {
        LOGI("LedService JNI: mokoid_setOff() is invoked.");
        if (sLedDevice == NULL) {
            LOGI("LedService JNI: sLedDevice was not fetched correctly.");
            return -1;
        } else {
            return sLedDevice->set_on(sLedDevice, led);     //调用 HAL 层的注册方法
        }
}
/** helper APIs——JNI 通过 LED_HARDSOFTWARE_MODULE_ID 找到对应的 stub*/
static inline int led_control_open(const struct hw_module_t* module,
            struct led_control_device_t** device) {
        return module->methods->open(module,
                LED_HARDWARE_MODULE_ID, (struct hw_device_t**)device);
}
static jboolean
mokoid_init(JNIEnv *env, jclass clazz)
{
        led_module_t* module;
        if (hw_get_module(LED_HARDWARE_MODULE_ID, (const hw_module_t**)&module) == 0) {
            LOGI("LedService JNI: LED Stub found.");
            if (led_control_open(&module->common, &sLedDevice) == 0) {
             LOGI("LedService JNI: Got Stub operations.");
                return 0;
            }
        }
        LOGE("LedService JNI: Get Stub operations failed.");
        return -1;
}
//JNINativeMethod 是 JNI 层的注册方法
static const JNINativeMethod gMethods[] = {
        {"_init", "()Z",//Framework 层调用_init 时触发
                    (void*)mokoid_init},
        { "_set_on", "(I)Z",
                            (void*)mokoid_setOn },
        { "_set_off", "(I)Z",
                            (void*)mokoid_setOff },
};
static int registerMethods(JNIEnv* env) {
        static const char* const kClassName =
            "com/mokoid/server/LedService";
        jclass clazz;
        /*  寻找类 class */
        clazz = env->FindClass(kClassName);
        if (clazz == NULL) {
```

```
        LOGE("Can't find class %s\n", kClassName);
        return -1;
    }
    /* register all the methods */
    if (env->RegisterNatives(clazz, gMethods,
            sizeof(gMethods) / sizeof(gMethods[0])) != JNI_OK)
    {
        LOGE("Failed registering methods for %s\n", kClassName);
        return -1;
    }
    return 0;
}
// ------------------------------------------------------------------------
//Framework 层加载 JNI 库时调用
jint JNI_OnLoad(JavaVM* vm, void* reserved) {
    JNIEnv* env = NULL;
    jint result = -1;
    if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        LOGE("ERROR: GetEnv failed\n");
        goto bail;
    }
    assert(env != NULL);
    if (registerMethods(env) != 0) {
        LOGE("ERROR: PlatformLibrary native registration failed\n");
        goto bail;
    }
    /* success -- return valid version number */
    result = JNI_VERSION_1_4;
bail:
    return result;
}
```

（3）Service 的实现代码

这里的 Service 属于 Framework 层，实现文件是 LedService.java，保存在目录 Frameworks/base/service/java/com/mokoid/server 中。

LedService.java 的具体实现代码如下所示。

```
package com.mokoid.server;

import android.util.Config;
import android.util.Log;
import android.content.Context;
import android.os.Binder;
import android.os.Bundle;
import android.os.RemoteException;
import android.os.IBinder;
import mokoid.hardware.ILedService;

public final class LedService extends ILedService.Stub {
```

```
    static {
        System.load("/system/lib/libmokoid_runtime.so");       //加载 JNI 动态库
    }

    public LedService() {
        Log.i("LedService", "Go to get LED Stub...");
     _init();
    }

    /*
     * Mokoid LED  本地方法
     */
    public boolean setOn(int led) {
        Log.i("MokoidPlatform", "LED On");
     return _set_on(led);
    }
    public boolean setOff(int led) {
        Log.i("MokoidPlatform", "LED Off");
     return _set_off(led);
    }
    private static native boolean _init();                    //声明 JNI 库可以提供的方法
    private static native boolean _set_on(int led);
    private static native boolean _set_off(int led);
}
```

（4）编写测试应用程序

测试应用程序属于 APP 层，文件 apps/LedClient/src/com/mokoid/LedClient /LedClient.java 的实现代码如下所示。

```
package com.mokoid.LedClient;
import com.mokoid.server.LedService;                          //导入 Framework 层的 LedService

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class LedClient extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

     LedService ls = new LedService();                        //实例化 LedService
     ls.setOn(1);                                             //通过 LedService 提供的方法控制底层硬件

        TextView tv = new TextView(this);
        tv.setText("LED 0 is on.");
        setContentView(tv);
    }
}
```

4.6.3　通过 Manager 调用 service 的实现代码

（1）实现 Manager

应用程序通过 Manager 和 Service 实现通信功能，文件 frameworks/base/core/java/mokoid/hardware/LedManager.java 的实现代码如下所示。

```
package mokoid.hardware;

import android.content.Context;
import android.os.Binder;
import android.os.Bundle;
import android.os.Parcelable;
import android.os.ParcelFileDescriptor;
import android.os.Process;
import android.os.RemoteException;
import android.os.Handler;
import android.os.Message;
import android.os.ServiceManager;
import android.util.Log;
import mokoid.hardware.ILedService;

public class LedManager
{
    private static final String TAG = "LedManager";
    private ILedService mLedService;

    public LedManager() {

        mLedService = ILedService.Stub.asInterface(
                                ServiceManager.getService("led"));

    if (mLedService != null) {
            Log.i(TAG, "The LedManager object is ready.");
    }
    }

    public boolean LedOn(int n) {
        boolean result = false;

        try {
            result = mLedService.setOn(n);
        } catch (RemoteException e) {
            Log.e(TAG, "RemoteException in LedManager.LedOn:", e);
        }
        return result;
    }
```

```
    public boolean LedOff(int n) {
        boolean result = false;

        try {
            result = mLedService.setOff(n);
        } catch (RemoteException e) {
            Log.e(TAG, "RemoteException in LedManager.LedOff:", e);
        }
        return result;
    }
}
```

因为LedService和LedManager分别属于不同的进程，所以在此需要考虑不同进程之间的通信问题。此时在 Manager 中可以增加一个 AIDL 文件来描述通信接口，文件 frameworks/base/core/java/mokoid/hardware/ILedService.aidl 的实现代码如下所示。

```
package mokoid.hardware;

interface ILedService
{
    boolean setOn(int led);
    boolean setOff(int led);
}
```

（2）实现 SystemServer

SystemServer 属于 APP 层，文件 apps/LedTest/src/com/mokoid/LedTest/LedSystemServer.java 的主要实现代码如下所示。

```
package com.mokoid.LedTest;

import com.mokoid.server.LedService;

import android.os.IBinder;
import android.os.ServiceManager;
import android.util.Log;
import android.app.Service;
import android.content.Context;
import android.content.Intent;

public class LedSystemServer extends Service {
    //代表一个后台进程
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
    public void onStart(Intent intent, int startId) {
        Log.i("LedSystemServer", "Start LedService...");
    /* Please also see SystemServer.java for your interests. */
    LedService ls = new LedService();
```

```
            try {
                ServiceManager.addService("led", ls);
            } catch (RuntimeException e) {
                Log.e("LedSystemServer", "Start LedService failed.");
            }
        }
}
```

（3）APP 测试程序

此处的测试程序属于 APP 层，文件 mokoid-read-only/apps/LedTest/src/com /mokoid/LedTest/LedTest.java 的实现代码如下所示。

```
package com.mokoid.LedTest;
import mokoid.hardware.LedManager;
import com.mokoid.server.LedService;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.widget.TextView;
import android.widget.Button;
import android.content.Intent;
import android.view.View;

public class LedTest extends Activity implements View.OnClickListener {
    private LedManager mLedManager = null;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        startService(new Intent("com.mokoid.systemserver"));
        Button btn = new Button(this);
        btn.setText("Click to turn LED 1 On");
     btn.setOnClickListener(this);
        setContentView(btn);
    }
    public void onClick(View v) {
        if (mLedManager == null) {
        Log.i("LedTest", "Creat a new LedManager object.");
        mLedManager = new LedManager();
        }
        if (mLedManager != null) {
        Log.i("LedTest", "Got LedManager object.");
    }
        mLedManager.LedOn(1);
        TextView tv = new TextView(this);
        tv.setText("LED 1 is On.");
        setContentView(tv);
    }
}
```

4.7　HAL 和系统移植

Android 的硬件抽象层和系统移植密切相关，本节将详细讲解移植 Android 5.0 HAL 的基本知识，为读者学习本书后面的知识打下基础。

4.7.1　移植各个 Android 部件的方式

在 Android 系统中，不同子系统的移植方法不同。不同部件的移植方式如下所示。

☑ 显示系统：使用 Framebuffer 标准或其他驱动程序，对应的硬件抽象层是 Gralloc。
☑ 用户输入系统：使用 Event 设备的驱动程序，对应的硬件抽象层是 EventHub。
☑ 3D 加速系统：使用非标准的驱动程序，对应的硬件抽象层是 OpenGL。
☑ 音频系统：使用非标准的驱动程序，对应的是 C++继承的硬件抽象层。
☑ 视频输出系统：使用非标准的驱动程序，对应的硬件抽象层是 overlay 模块。
☑ 摄像头系统：使用非标准的驱动程序，对应的是 C++继承的硬件抽象层。
☑ 多媒体解码系统：使用非标准的驱动程序，对应的硬件抽象层是 Skia 和 OpenMax 插件。
☑ 电话系统：使用非标准的驱动程序，对应的硬件抽象层是动态开发插件库。
☑ GPS 定位系统：使用非标准的驱动程序，对应的硬件抽象层通常是直接接口。
☑ 无线局域网：使用 WLAN 驱动程序，对应的硬件抽象层分别是 Linux 下的 WPA 和 Android 下的 Wi-Fi。
☑ 蓝牙系统：使用 Bluetooth 驱动程序，对应的硬件抽象层分别是 Linux 下的 Bluez 和 Android 下的 Bluedroid。
☑ 传感器系统：使用非标准的驱动程序，对应的硬件抽象层是 Sensor 硬件模块。
☑ 振动器系统：使用 Sys 文件系统中固定位置的驱动程序，对应的硬件抽象层是 Android 标准的直接接口。
☑ 背光和指示灯系统：使用非标准的驱动程序，对应的硬件抽象层是 Light 硬件模块。
☑ 警告器系统：使用 Misc 驱动程序，对应的硬件抽象层是 Android 标准的 JNI 层。
☑ 电池管理系统：使用 Sys 文件系统中固定位置的驱动程序，对应的硬件抽象层是 Android 标准的直接接口。

4.7.2　设置设备权限

当 Android 系统启动时，在内核引导参数上一般都会设置 init=/init，此时如果内核成功挂载了这个文件系统，首先运行的就是这个根目录下的 init 程序。该 init 程序是 Android 系统运行后的第一个用户空间的程序，以守护进程的方式运行。

当需要增加驱动程序的设备节点时，需随之更改这些设备节点的属性，这些更改内容被保存在文件 system/core/init/devices.c 中。此文件代码比较冗长，接下来将只对和权限有关的代码进行讲解。

☑ 定义 perms_表示设备的类型，具体代码如下所示。



```
struct perms_ {
	char *name;
	mode_t perm;
	unsigned int uid;
	unsigned int gid;
	unsigned short prefix;
};
```

☑ 定义数组 devperms 表示系统中的设备，具体代码如下所示。

```
static struct perms_ devperms[] = {
	{ "/dev/null", 0666, AID_ROOT, AID_ROOT, 0 },
	{ "/dev/zero", 0666, AID_ROOT, AID_ROOT, 0 },
	{ "/dev/full", 0666, AID_ROOT, AID_ROOT, 0 },
	{ "/dev/ptmx", 0666, AID_ROOT, AID_ROOT, 0 },
	{ "/dev/tty", 0666, AID_ROOT, AID_ROOT, 0 },
	{ "/dev/random", 0666, AID_ROOT, AID_ROOT, 0 },
	{ "/dev/urandom", 0666, AID_ROOT, AID_ROOT, 0 },
	{ "/dev/ashmem", 0666, AID_ROOT, AID_ROOT, 0 },
	{ "/dev/binder", 0666, AID_ROOT, AID_ROOT, 0 },

		/* logger should be world writable (for logging) but not readable */
	{ "/dev/log/", 0662, AID_ROOT, AID_LOG, 1 },

		/* these should not be world writable */
	{ "/dev/android_adb", 0660, AID_ADB, AID_ADB, 0 },
	{ "/dev/android_adb_enable", 0660, AID_ADB, AID_ADB, 0 },
	{ "/dev/ttyMSM0", 0600, AID_BLUETOOTH, AID_BLUETOOTH, 0 },
	{ "/dev/ttyHS0", 0600, AID_BLUETOOTH, AID_BLUETOOTH, 0 },
	{ "/dev/uinput", 0600, AID_BLUETOOTH, AID_BLUETOOTH, 0 },
	{ "/dev/alarm", 0664, AID_SYSTEM, AID_RADIO, 0 },
	{ "/dev/tty0", 0660, AID_ROOT, AID_SYSTEM, 0 },
	{ "/dev/graphics/", 0660, AID_ROOT, AID_GRAPHICS, 1 },
	{ "/dev/hw3d", 0660, AID_SYSTEM, AID_GRAPHICS, 0 },
	{ "/dev/input/", 0660, AID_ROOT, AID_INPUT, 1 },
	{ "/dev/eac", 0660, AID_ROOT, AID_AUDIO, 0 },
	{ "/dev/cam", 0660, AID_ROOT, AID_CAMERA, 0 },
	{ "/dev/pmem", 0660, AID_SYSTEM, AID_GRAPHICS, 0 },
	{ "/dev/pmem_gpu", 0660, AID_SYSTEM, AID_GRAPHICS, 1 },
	{ "/dev/pmem_adsp", 0660, AID_SYSTEM, AID_AUDIO, 1 },
	{ "/dev/pmem_camera", 0660, AID_SYSTEM, AID_CAMERA, 1 },
	{ "/dev/oncrpc/", 0660, AID_ROOT, AID_SYSTEM, 1 },
	{ "/dev/adsp/", 0660, AID_SYSTEM, AID_AUDIO, 1 },
	{ "/dev/mt9t013", 0660, AID_SYSTEM, AID_SYSTEM, 0 },
	{ "/dev/akm8976_daemon",0640, AID_COMPASS, AID_SYSTEM, 0 },
	{ "/dev/akm8976_aot", 0640, AID_COMPASS, AID_SYSTEM, 0 },
	{ "/dev/akm8976_pffd", 0640, AID_COMPASS, AID_SYSTEM, 0 },
	{ "/dev/msm_pcm_out", 0660, AID_SYSTEM, AID_AUDIO, 1 },
	{ "/dev/msm_pcm_in", 0660, AID_SYSTEM, AID_AUDIO, 1 },
	{ "/dev/msm_pcm_ctl", 0660, AID_SYSTEM, AID_AUDIO, 1 },
	{ "/dev/msm_snd", 0660, AID_SYSTEM, AID_AUDIO, 1 },
```

```
	{ "/dev/msm_mp3", 0660, AID_SYSTEM, AID_AUDIO, 1 },
	{ "/dev/msm_audpre", 0660, AID_SYSTEM, AID_AUDIO, 0 },
	{ "/dev/htc-acoustic", 0660, AID_SYSTEM, AID_AUDIO, 0 },
	{ "/dev/smd0", 0640, AID_RADIO, AID_RADIO, 0 },
	{ "/dev/qmi", 0640, AID_RADIO, AID_RADIO, 0 },
	{ "/dev/qmi0", 0640, AID_RADIO, AID_RADIO, 0 },
	{ "/dev/qmi1", 0640, AID_RADIO,   AID_RADIO, 0 },
	{ "/dev/qmi2", 0640, AID_RADIO, AID_RADIO, 0 },
	{ NULL, 0, 0, 0, 0 },
};
```

在上述数组中分别设置了设备的权限、所属用户和所属组，各个权限值的含义和 Linux 中的完全一致。3 个数组分别表示所属用户、所属组和其他用户的权限，其中，4 表示可读，2 表示可写，1 表示可执行。例如，数组内的首行代码如下所示。

```
{ "/dev/null", 0666, AID_ROOT, AID_ROOT, 0 },
```

/dev/null 是一个标准的设备，其权限是 0666，表示任何用户可以对其进行读写操作。如果需要增加一个新的设备节点文件，需要在数组 devperms 中新增加一行内容。

☑ 两个函数。在文件中有两个比较重要的函数：handle_device_event()和 make_device()，具体代码如下所示。

```
static void handle_device_event(struct uevent *uevent)
{
	…
	/* are we block or char? where should we live? */
	if(!strncmp(uevent->path, "/block", 6)) {
		block = 1;
		base = "/dev/block/";				//根据 uevent 路径改变该节点路径
		mkdir(base, 0755);
	} else {
		block = 0;
			/* this should probably be configurable somehow */
		if(!strncmp(uevent->path, "/class/graphics/", 16)) {
			base = "/dev/graphics/";		//根据 uevent 路径改变该 uevent 需要创建节点的路径
			mkdir(base, 0755);
		} else if (!strncmp(uevent->path, "/class/oncrpc/", 14)) {
			base = "/dev/oncrpc/";
			mkdir(base, 0755);
		} else if (!strncmp(uevent->path, "/class/adsp/", 12)) {
			base = "/dev/adsp/";
			mkdir(base, 0755);
		} else if(!strncmp(uevent->path, "/class/input/", 13)) {
			base = "/dev/input/";			//根据 uevent 路径改变该 uevent 需要创建节点的路径
			mkdir(base, 0755);
		} else if(!strncmp(uevent->path, "/class/sensors/", 15)) {
			base = "/dev/sensors/";
			mkdir(base, 0755);
		} else if(!strncmp(uevent->path, "/class/mtd/", 11)) {
			base = "/dev/mtd/";			//根据 uevent 路径改变该 uevent 需要创建节点的路径
```

```
                mkdir(base, 0755);
            } else if(!strncmp(uevent->path, "/class/misc/", 12) &&
                        !strncmp(name, "log_", 4)) {
                base = "/dev/log/";           //根据 uevent 路径改变该 uevent 需要创建节点的路径
                mkdir(base, 0755);
                name += 4;
            } else if(!strncmp(uevent->path, "/class/sound/", 13)) {
                base = "/dev/snd/";
                mkdir(base, 0755);
            } else
                base = "/dev/";
        }

        snprintf(devpath, sizeof(devpath), "%s%s", base, name);

        if(!strcmp(uevent->action, "add")) {
            make_device(devpath, block, uevent->major, uevent->minor);//创建节点文件文件 devpath
            return;
        }
        …
}
static void make_device(const char *path, int block, int major, int minor)
{
        unsigned uid;
        unsigned gid;
        mode_t mode;
        dev_t dev;
        if(major > 255 || minor > 255)
            return;
        mode = get_device_perm(path, &uid, &gid) | (block ? S_IFBLK : S_IFCHR);//获取将要创建的节点是否需要
重设其 mode 数值
        dev = (major << 8) | minor;
        mknod(path, mode, dev);
        chown(path, uid, gid);
}
```

函数 get_device_perm()的功能是验证路径 path 是否和 devperms[]数组中的 inode 路径相同，如果相同则返回 devperms[]数组中指定的 uid、gid 和 mode 数值。这样 make_device 会向/dev 创建 inode 节点，并同时改变该 inode 的 uid 和 gid.[luther.gliethtt]。

☑ 和用户名相关的名称。

在文件 system/core/include/private/android_filesystem_config.h 中定义了和用户名相关的名称，其中，android_id_info 表示用户名 id 的属性。android_id_info 的定义代码如下所示。

```
struct android_id_info {
        const char *name;
        unsigned aid;
};
```

各个用户名 id 被定义在数组 android_ids[]中，此数组表示了一个映射关系，能够将字符串和整数

值对应起来。android_ids[]的定义代码如下所示。

```
static struct android_id_info android_ids[] = {
	{ "root", AID_ROOT, },
	{ "system", AID_SYSTEM, },
	{ "radio", AID_RADIO, },
	{ "bluetooth", AID_BLUETOOTH, },
	{ "graphics", AID_GRAPHICS, },
	{ "input", AID_INPUT, },
	{ "audio", AID_AUDIO, },
	{ "camera", AID_CAMERA, },
	{ "log", AID_LOG, },
	{ "compass", AID_COMPASS, },
	{ "mount", AID_MOUNT, },
	{ "wifi", AID_WIFI, },
	{ "dhcp", AID_DHCP, },
	{ "adb", AID_ADB, },
	{ "install", AID_INSTALL, },
	{ "media", AID_MEDIA, },
	{ "shell", AID_SHELL, },
	{ "cache", AID_CACHE, },
	{ "diag", AID_DIAG, },
	{ "net_bt_admin", AID_NET_BT_ADMIN, },
	{ "net_bt", AID_NET_BT, },
	{ "inet", AID_INET, },
	{ "net_raw", AID_NET_RAW, },
	{ "misc", AID_MISC, },
	{ "nobody", AID_NOBODY, },
};
```

4.7.3 init.rc 初始化

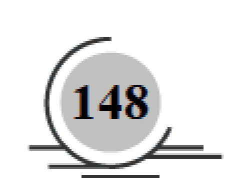
文件 system/core/rootdir/init.rc 可以实现一些简单的初始化操作，Android 5.0 中的启动脚本文件是 init.rc。init.rc 脚本被直接安装到目标系统的根目录下，并被 init 可执行的程序解析。

在 Android 5.0 中，init.rc 是在 init 启动后被执行的启动脚本，主要包含如下内容。

- ☑ Commands：命令。
- ☑ Actions：动作。
- ☑ Triggers：触发条件。
- ☑ Services：服务。
- ☑ Options：选项。
- ☑ Propertise：属性。

4.7.4 文件系统的属性

在文件 system/core/include/private/android_filesystem_config.h 中定义了各个文件的属性，其中 fs_path_config 表示文件系统路径的属性，具体定义代码如下所示。



```
struct fs_path_config {
    unsigned mode;              //模式
    unsigned uid;               //用户 id
    unsigned gid;               //组 id
    const char *prefix;         //目录前缀
};
```

在数组 android_dirs[]中定义了子目录的属性，定义代码如下所示。

```
static struct fs_path_config android_dirs[] = {
    { 00770, AID_SYSTEM, AID_CACHE,  "cache" },
    { 00771, AID_SYSTEM, AID_SYSTEM, "data/app" },
    { 00771, AID_SYSTEM, AID_SYSTEM, "data/app-private" },
    { 00771, AID_SYSTEM, AID_SYSTEM, "data/dalvik-cache" },
    { 00771, AID_SYSTEM, AID_SYSTEM, "data/data" },
    { 00771, AID_SHELL, AID_SHELL, "data/local/tmp" },
    { 00771, AID_SHELL, AID_SHELL, "data/local" },
    { 01771, AID_SYSTEM, AID_MISC, "data/misc" },
    { 00770, AID_DHCP, AID_DHCP, "data/misc/dhcp" },
    { 00771, AID_SYSTEM, AID_SYSTEM, "data" },
    { 00750, AID_ROOT, AID_SHELL, "sbin" },
    { 00755, AID_ROOT, AID_SHELL, "system/bin" },
    { 00755, AID_ROOT, AID_SHELL, "system/xbin" },
    { 00777, AID_ROOT, AID_ROOT, "system/etc/ppp" }, /* REMOVE */
    { 00777, AID_ROOT, AID_ROOT, "sdcard" },
    { 00755, AID_ROOT, AID_ROOT, 0 },
};
```

在数组 android_files[]中定义默认文件的属性，定义代码如下所示。

```
static struct fs_path_config android_files[] = {
    { 00555, AID_ROOT, AID_ROOT, "system/etc/ppp/ip-up" },
    { 00555, AID_ROOT, AID_ROOT, "system/etc/ppp/ip-down" },
    { 00440, AID_ROOT, AID_SHELL, "system/etc/init.goldfish.rc" },
    { 00550, AID_ROOT, AID_SHELL, "system/etc/init.goldfish.sh" },
    { 00440, AID_ROOT, AID_SHELL, "system/etc/init.trout.rc" },
    { 00550, AID_ROOT, AID_SHELL, "system/etc/init.ril" },
    { 00550, AID_ROOT, AID_SHELL, "system/etc/init.testmenu" },
    { 00550, AID_ROOT, AID_SHELL, "system/etc/init.gprs-pppd" },
    { 00550, AID_DHCP, AID_SHELL, "system/etc/dhcpcd/dhcpcd-run-hooks" },
    { 00440, AID_BLUETOOTH, AID_BLUETOOTH, "system/etc/dbus.conf" },
    { 00440, AID_BLUETOOTH, AID_BLUETOOTH, "system/etc/bluez/hcid.conf" },
    { 00440, AID_BLUETOOTH, AID_BLUETOOTH, "system/etc/bluez/input.conf" },
    { 00440, AID_BLUETOOTH, AID_BLUETOOTH, "system/etc/bluez/audio.conf" },
    { 00440, AID_RADIO, AID_AUDIO, "/system/etc/AudioPara4.csv" },
    { 00644, AID_SYSTEM, AID_SYSTEM, "data/app/*" },
    { 00644, AID_SYSTEM, AID_SYSTEM, "data/app-private/*" },
    { 00644, AID_APP, AID_APP, "data/data/*" },
        /* the following two files are INTENTIONALLY set-gid and not set-uid.
         * Do not change */
    { 02755, AID_ROOT, AID_NET_RAW, "system/bin/ping" },
```

```
        { 02755, AID_ROOT, AID_INET, "system/bin/netcfg" },
         /* the following four files are INTENTIONALLY set-uid, but they
           * are NOT included on user builds */
        { 06755, AID_ROOT, AID_ROOT, "system/xbin/su" },
        { 06755, AID_ROOT, AID_ROOT, "system/xbin/librank" },
        { 06755, AID_ROOT, AID_ROOT, "system/xbin/procrank" },
        { 06755, AID_ROOT, AID_ROOT, "system/xbin/procmem" },
        { 00755, AID_ROOT, AID_SHELL, "system/bin/*" },
        { 00755, AID_ROOT, AID_SHELL, "system/xbin/*" },
        { 00750, AID_ROOT, AID_SHELL, "sbin/*" },
        { 00755, AID_ROOT, AID_ROOT, "bin/*" },
        { 00750, AID_ROOT, AID_SHELL, "init*" },
        { 00644, AID_ROOT, AID_ROOT, 0 },
};
```

4.8　开发自己的 HAL

经过本章前面内容的学习，已经了解了 HAL 的具体架构和运行原理。本节将简单演示开发自己的 HAL 的具体过程，并讲解编译调用的方法。

4.8.1　封装 HAL 接口

可以将 Android 的硬件驱动程序看作在内核层，HAL 负责封装硬件驱动，然后再经过 JNI 接口的封装才能给 Java 应用程序调用。封装 HAL 层接口的具体流程如下所示。

（1）在 hardware/libhardware/include/hardware 目录下添加头文件 hello.h，具体方法可以参考当前目录下的头文件 overlay.h。头文件 hello.h 的具体代码如下所示。

```
/****************************************************
*android_hal_hello_demo
*hello.h
****************************************************/
#ifndef ANDROID_HELLO_INTERFACE_H
#define ANDROID_HELLO_INTERFACE_H
#include <hardware/hardware.h>
__BEGIN_DECLS
#define HELLO_HARDWARE_MODULE_ID "hello"
struct hello_module_t {
struct hw_module_t common;
};
struct hello_device_t {
struct hw_device_t common;
int fd;
int (*get_val)(struct hello_device_t *dev,int val);
int (*set_val)(struct hello_device_t *dev,int val);
};
__END_DECLS;
#endif
```

（2）在 hardware/libhardware/modules 目录下新建一个名为 hello 的文件夹，并在此文件夹中新建文件 hello.c 和 Android.mk 文件，具体方法读者可以参考 modules/overlay 目录下的内容。文件 hello.c 的具体代码如下所示。

```
/*********************************************************************************
*android_hal_hello_demo
*hello.c
*********************************************************************************/
#include <hardware/hardware.h>
#include <hardware/hello.h>
#include <fcntl.h>
#include <errno.h>
#include <cutils/log.h>
#include <cutils/atomic.h>
#define LOG_TAG "hello_stub"
#define DEVICE_NAME "/dev/hello"
#define MODULE_NAME "Hello"
#define MODULE_AUTHOR "729017304@qq.com"

static int hello_device_open(const struct hw_module_t *module,const char *name,struct hw_device_t** device);
static int hello_device_close(struct hw_device_t* device);
static int hello_set_val(struct hello_device_t*dev,int val);
static int hello_get_val(struct hello_device_t *dev,int *val);
static struct hw_module_methods_t hello_module_methods    = {
open : hello_device_open
};

struct hello_module_t HAL_MODULE_INFO_SYM = {
common: {
tag: HARDWARE_MODULE_TAG,
version_major: 1,
version_minor: 0,

id: HELLO_HARDWARE_MODULE_ID,
name: MODULE_NAME,
author: MODULE_AUTHOR,
methods: &hello_module_methods,
}
};

static int hello_device_open(const struct hw_module_t* module, const char* name, struct hw_device_t** device)
{
struct hello_device_t* dev;
dev = (struct hello_device_t*)malloc(sizeof(struct hello_device_t));
if(!dev) {
LOGE("Hello Stub: failed to alloc space");
return -EFAULT;
}
memset(dev, 0, sizeof(struct hello_device_t));
dev->common.tag = HARDWARE_DEVICE_TAG;
```

```
dev->common.version = 0;
dev->common.module = (hw_module_t*)module;
dev->common.close = hello_device_close;
dev->set_val = hello_set_val;
dev->get_val = hello_get_val;
if((dev->fd = open(DEVICE_NAME, O_RDWR)) == -1) {
LOGE("Hello Stub: failed to open /dev/hello -- %s.", strerror(errno));free(dev);
return -EFAULT;
}
*device = &(dev->common);
LOGI("Hello Stub: open /dev/hello successfully.");
return 0;
}
static int hello_device_close(struct hw_device_t* device) {
struct hello_device_t* hello_device = (struct hello_device_t*)device;
if(hello_device) {
close(hello_device->fd);
free(hello_device);
}
return 0;
}
static int hello_set_val(struct hello_device_t* dev, int val) {
LOGI("Hello Stub: set value %d to device.", val);
write(dev->fd, &val, sizeof(val));
return 0;
}
static int hello_get_val(struct hello_device_t* dev, int* val) {
if(!val) {
LOGE("Hello Stub: error val pointer");
return -EFAULT;
}
read(dev->fd, val, sizeof(*val));
LOGI("Hello Stub: get value %d from device", *val);
return 0;
}
```

Android.mk 文件的具体代码如下所示。

```
/*********************************************************************************
*android_hal_hello_demo
*Android.mk
*********************************************************************************/
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_PRELINK_MODULE := false
LOCAL_MODULE_PATH := $(TARGET_OUT_SHARED_LIBRARIES)/hw
LOCAL_SHARED_LIBRARIES := liblog
LOCAL_SRC_FILES := hello.c
LOCAL_MODULE := hello.default
include $(BUILD_SHARED_LIBRARY)
```

4.8.2　开始编译

因为在获取源码后已经 make 编译过一次 Android 源码了，所以此时不需要重新 make clean 并编译了，只需将模块编译好，然后再执行 make snod 命令即可将新的模块编译到镜像中。在 Android 源码的 build 目录下有一个配置环境的脚本文件 envsetup.sh，在此文件中包含了编译工具 m、mm 和 mmm。在此使用工具 mmm 进行编译。

（1）在 Android 源码包中执行如下所示的命令。

```
[root@localhost Android-4.4]# sh build/envsetup.sh
[root@localhost Android-4.4]#croot
```

（2）使用 mmm 工具编译模块，具体命令如下所示。

```
[root@localhost Android-4.4]#mmm hardware/libhardware/modules/hello
```

如果出现找不到 liblog.so 库文件的错误，则需要编译生成 liblog.so 这个库文件来解决。编译界面效果如图 4-7 所示。

```
[root@localhost Android-4.4]# mmm hardware/libhardware/modules/hello
============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=4.4
TARGET_PRODUCT=generic
TARGET_BUILD_VARIANT=eng
TARGET_SIMULATOR=
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
HOST_ARCH=x86
HOST_OS=linux
HOST_BUILD_TYPE=release
BUILD_ID=GINGERBREAD
============================================
make: 进入目录"/home/linux_kernel/tiny210/linux_kernel_sources/Android-4.4"
target thumb C: hello.default <= hardware/libhardware/modules/hello/hello.c
hardware/libhardware/modules/hello/hello.c: In function 'hello_device_open':
hardware/libhardware/modules/hello/hello.c:49: warning: assignment from incompatible pointer type
make: *** 没有规则可以创建"out/target/product/generic/obj/SHARED_LIBRARIES/hello.default_intermediates/LINKED/hello.default.so
"需要的目标"out/target/product/generic/obj/lib/liblog.so"。 停止。
make: 离开目录"/home/linux_kernel/tiny210/linux_kernel_sources/Android-4.4"
```

图 4-7　编译界面

（3）编译生成 liblog.so，具体命令如下所示。

```
[root@localhost Android-4.4]#make liblog
```

编译界面效果如图 4-8 所示。

```
文件(F)  编辑(E)  查看(V)  搜索(S)  终端(T)  帮助(H)
target arm C: libm <= bionic/libm/src/s_scalbnf.c
bionic/libm/src/s_scalbnf.c: In function 'scalbnf':
bionic/libm/src/s_scalbnf.c:49: warning: suggest explicit braces to avoid ambiguous 'else'
bionic/libm/src/s_scalbnf.c: At top level:
bionic/libm/src/s_scalbnf.c:58: warning: data definition has no type or storage class
bionic/libm/src/s_scalbnf.c:58: warning: type defaults to 'int' in declaration of '__strong_reference'
bionic/libm/src/s_scalbnf.c:58: warning: parameter names (without types) in function declaration
target SharedLib: libm (out/target/product/generic/obj/SHARED_LIBRARIES/libm_intermediates/LINKED/libm.so)
target Prelink: libm (out/target/product/generic/symbols/system/lib/libm.so)
libelfcopy: Warning: Range lists in .debug_info section aren't in ascending order!
target Strip: libm (out/target/product/generic/obj/lib/libm.so)
target SharedLib: liblog (out/target/product/generic/obj/SHARED_LIBRARIES/liblog_intermediates/LINKED/liblog.so)
target Non-prelinked: liblog (out/target/product/generic/symbols/system/lib/liblog.so)
target Strip: liblog (out/target/product/generic/obj/lib/liblog.so)
Notice file: system/core/liblog/NOTICE -- out/target/product/generic/obj/NOTICE_FILES/src//system/lib/liblog.so.txt
Notice file: bionic/libc/NOTICE -- out/target/product/generic/obj/NOTICE_FILES/src//system/lib/libc.so.txt
Notice file: bionic/libdl/NOTICE -- out/target/product/generic/obj/NOTICE_FILES/src//system/lib/libdl.so.txt
Install: out/target/product/generic/system/lib/libdl.so
Notice file: bionic/libc/NOTICE -- out/target/product/generic/obj/NOTICE_FILES/src//system/lib/libc_common.a.txt
Install: out/target/product/generic/system/lib/libc.so
Notice file: bionic/libstdc++/NOTICE -- out/target/product/generic/obj/NOTICE_FILES/src//system/lib/libstdc++.so.txt
Install: out/target/product/generic/system/lib/libstdc++.so
Notice file: bionic/libm/NOTICE -- out/target/product/generic/obj/NOTICE_FILES/src//system/lib/libm.so.txt
Install: out/target/product/generic/system/lib/libm.so
Notice file: system/core/liblog/NOTICE -- out/target/product/generic/obj/NOTICE_FILES/src//system/lib/liblog.a.txt
Install: out/target/product/generic/system/lib/liblog.so
```

图 4-8　编译界面

（4）接下来开始重新编译，具体命令如下所示。

```
[root@localhost Android-4.4]#mmm hardware/libhardware/modules/hello
```

最终会成功生成库文件 hello.default.so，如图 4-9 所示。

```
[root@localhost Android-4.4]# mmm hardware/libhardware/modules/hello
============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=4.4
TARGET_PRODUCT=generic
TARGET_BUILD_VARIANT=eng
TARGET_SIMULATOR=
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
HOST_ARCH=x86
HOST_OS=linux
HOST_BUILD_TYPE=release
BUILD_ID=GINGERBREAD
============================================
make: 进入目录“/home/linux_kernel/tiny210/linux_kernel_sources/Android-4.4”
target SharedLib: hello.default (out/target/product/generic/obj/SHARED_LIBRARIES/hello.default_intermediates/LINKED/hello.defa
ult.so)
target Non-prelinked: hello.default (out/target/product/generic/symbols/system/lib/hw/hello.default.so)
target Strip: hello.default (out/target/product/generic/obj/lib/hello.default.so)
Install: out/target/product/generic/system/lib/hw/hello.default.so
make: 离开目录“/home/linux_kernel/tiny210/linux_kernel_sources/Android-4.4”
[root@localhost Android-4.4]#
```

图 4-9　成功编译界面

（5）重新打包镜像，具体命令如下所示。

```
[root@localhost Android-4.4]make snod
```

（6）最后的工作是重新封装 JNI，这就不是本章的内容了。读者可以参阅本书前面 JNI 章节中的 Android 驱动使用 JNI 调用的知识，也可以在网络中搜索相关资料。

第 5 章　Binder 通信机制详解

在 Android 系统中，应用程序都是由 Activity 和 Service 组成的。Service 通常运行在独立的进程中，而 Activity 既可能运行在同一个进程中，也可能运行在不同的进程中。不在同一个进程中的 Activity 或 Service 通过 Binder 机制是如何实现进程间的通信功能的呢？Binder 是 Android 系统提供的一种 IPC（进程间通信）机制。由于 Android 是基于 Linux 内核的，因此，除了 Binder 以外，还存在其他 IPC 机制，例如管道和 socket 等。Binder 相对于其他 IPC 机制来说，就更加灵活和方便了。Binder 的驱动代码在 kernel/drivers/staing/android/binder.c 目录中保存，另外该目录下还有一个 binder.h 头文件。Binder 是一个虚拟设备，所以它的代码相比而言还算简单，读者只要有基本的 Linux 驱动开发方面的知识就能读懂。/proc/binder 目录下的内容可用来查看 Binder 设备的运行状况。本章将详细讲解 Android 的进程通信机制 Binder 的基本源码，并详细分析 IPC 通信机制，为读者学习本书后面的知识打下基础。

5.1　分析 Binder 驱动程序

可以将 Android 系统看作是一个基于 Binder 通信的 C/S 架构，Binder 像网络一样把系统的各个部分连接在了一起。在基于 Binder 通信的 C/S 架构体系中，除了 C/S 架构所包括的 Client 端和 Server 端外，Android 系统还有一个全局的 ServiceManager 端，其作用是管理系统中的各种服务（Service）。

Binder 采用 AIDL（Android Interface Description Language）来描述进程间通信的接口。Binder 作为一个特殊的字符设备，其设备节点是/dev/binder。主要代码在文件 kernel/drivers/staging/binder.h 和 kernel/drivers/staging/binder.c 中实现。

本节将详细分析上述驱动文件的实现源码。

5.1.1　数据结构 binder_work

数据结构 binder_work 表示在 binder 驱动中进程所要处理的工作项，定义代码如下所示。

```
struct binder_work {
	struct list_head entry;
	enum {
		BINDER_WORK_TRANSACTION = 1,
		BINDER_WORK_TRANSACTION_COMPLETE,
		BINDER_WORK_NODE,
		BINDER_WORK_DEAD_BINDER,
		BINDER_WORK_DEAD_BINDER_AND_CLEAR,
		BINDER_WORK_CLEAR_DEATH_NOTIFICATION,
	} type;
};
```

在上述结构体定义中，entry 被定义为 list_head 类型，用于实现一个双向链表，能够存储所有 binder_

work 的队列，还包含了一个 enum 类型的 type：binder_work。

5.1.2 结构体 binder_node

结构体 binder_node 用来定义 Binder 实体对象。在 Android 系统中，每一个 Service 组件在 Binder 驱动程序中都有一个 Binder 实体对象。定义 binder_node 的代码如下所示。

```
struct binder_node {
        int debug_id;
        struct binder_work work;
        union {
                struct rb_node rb_node;
                struct hlist_node dead_node;
        };
        struct binder_proc *proc;
        struct hlist_head refs;
        int internal_strong_refs;
        int local_weak_refs;
        int local_strong_refs;
        void __user *ptr;
        void __user *cookie;
        unsigned has_strong_ref:1;
        unsigned pending_strong_ref:1;
        unsigned has_weak_ref:1;
        unsigned pending_weak_ref:1;
        unsigned has_async_transaction:1;
        unsigned accept_fds:1;
        unsigned min_priority:8;
        struct list_head async_todo;
};
```

驱动中的 Binder 实体也叫做“节点”，隶属于提供实体的进程。结构体 binder_node 中各个成员的具体说明如表 5-1 所示。

表 5-1　结构体 binder_node 中的成员说明信息

成　　员	含　　义
int debug_id;	用于调试
struct binder_work work;	当本节点引用计数发生改变，需要通知所属进程时，通过该成员挂入所属进程的 to-do 队列中，唤醒所属进程执行 Binder 实体引用计数的修改

union {
struct rb_node rb_node;
struct hlist_node dead_node;
};
	每个进程都维护一棵红黑树，以 Binder 实体在用户空间的指针，即本结构的 ptr 成员为索引存放该进程所有的 Binder 实体。这样驱动可以根据 Binder 实体在用户空间的指针很快找到其位于内核的节点。rb_node 用于将本节点链入该红黑树中
	销毁节点时需将 rb_node 从红黑树中摘除，但如果本节点还有引用没有切断，就用 dead_node 将节点隔离到另一个链表中，直到通知所有进程切断与该节点的引用后，该节点才可能被销毁

struct binder_proc *proc;	本成员指向节点所属的进程，即提供该节点的进程
struct hlist_head refs;	本成员是队列头，所有指向本节点的引用都链接在该队列中。这些引用可能隶属于不同的进程。通过该队列可以遍历指向该节点的所有引用

续表

成　员	含　义
int internal_strong_refs;	用以实现强指针的计数器：产生一个指向本节点的强引用，该计数就会加 1
int local_weak_refs;	驱动为传输中的 Binder 设置的弱引用计数。如果一个 Binder 打包在数据包中从一个进程发送到另一个进程，驱动会为该 Binder 增加引用计数，直到接收进程通过 BC_FREE_BUFFER 通知驱动释放该数据包的数据区为止
int local_strong_refs;	驱动为传输中的 Binder 设置的强引用计数
void __user *ptr;	指向用户空间 Binder 实体的指针，来自于 flat_binder_object 的 binder 成员
void __user *cookie;	指向用户空间的附加指针，来自于 flat_binder_object 的 cookie 成员
unsigned has_strong_ref; unsigned pending_strong_ref; unsigned has_weak_ref; unsigned pending_weak_ref;	这一组标志用于控制驱动与 Binder 实体所在进程交互式修改引用计数
unsigned has_async_transaction;	该成员表明该节点在 to-do 队列中有异步交互尚未完成。驱动将所有发送往接收端的数据包暂存在接收进程或线程开辟的 to-do 队列中。对于异步交互，驱动做了适当流控：如果 to-do 队列中有异步交互尚待处理则该成员置 1，这将导致新到的异步交互存放在本结构成员– asynch_todo 队列中，而不直接送到 to-do 队列中。目的是为同步交互让路，避免长时间阻塞发送端
unsigned accept_fds	表明节点是否同意接受文件方式的 Binder，来自 flat_binder_object 中 flags 成员的 FLAT_BINDER_FLAG_ACCEPTS_FDS 位。由于接收文件 Binder 会为进程自动打开一个文件，占用有限的文件描述符，节点可以设置该位拒绝这种行为
int min_priority	设置处理 Binder 请求的线程的最低优先级。发送线程将数据提交给接收线程处理时，驱动会将发送线程的优先级也赋予接收线程，使得数据即使跨了进程也能以同样优先级得到处理。不过如果发送线程优先级过低，接收线程将以预设的最小值运行 该域的值来自于 flat_binder_object 中的 flags 成员
struct list_head async_todo	异步交互等待队列；用于分流发往本节点的异步交互包

5.1.3　结构体 binder_ref

结构体 binder_ref 用来描述一个 Binder 引用对象，在 Android 系统中，每一个 Client 组件在 Binder 驱动程序中都有一个 Binder 引用对象。定义 binder_ref 的代码如下所示。

```
struct binder_ref {
        /* Lookups needed: */
        /*    node + proc => ref (transaction) */
        /*    desc + proc => ref (transaction, inc/dec ref) */
        /*    node => refs + procs (proc exit) */
        int debug_id;
        struct rb_node rb_node_desc;
        struct rb_node rb_node_node;
        struct hlist_node node_entry;
        struct binder_proc *proc;
        struct binder_node *node;
        uint32_t desc;
        int strong;
```

```
        int weak;
        struct binder_ref_death *death;
};
```

结构体 binder_ref 中各个成员的具体说明如表 5-2 所示。

表 5-2　结构体 binder_ref 中的成员说明信息

成　　员	含　　义
int debug_id;	调试用
struct rb_node rb_node_desc;	每个进程有一棵红黑树，进程所有引用以引用号（即本结构的 desc 域）为索引添入该树中。本成员用作链接到该树的一个节点
struct rb_node rb_node_node;	每个进程又有一棵红黑树，进程所有引用以节点实体在驱动中的内存地址（即本结构的 node 域）为索引添入该树中。本成员用作链接到该树的一个节点
struct hlist_node node_entry;	该域将本引用作为节点链入所指向的 Binder 实体结构 binder_node 中的 refs 队列
struct binder_proc *proc;	本引用所属的进程
struct binder_node *node;	本引用所指向的节点（Binder 实体）
uint32_t desc;	本结构的引用号
int strong;	强引用计数
int weak;	弱引用计数
struct binder_ref_death *death;	应用程序向驱动发送 BC_REQUEST_DEATH_NOTIFICATION 或 BC_CLEAR_DEATH_NOTIFICATION 命令从而当 Binder 实体销毁时能够收到来自驱动的提醒。该域不为空表明用户订阅了对应实体销毁的提醒

5.1.4　通知结构体 binder_ref_death

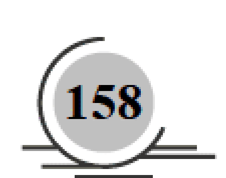
binder_ref_death 是一个通知结构体，只要某进程对某 binder 引用订阅了其实体的死亡通知，那么 binder 驱动将会为该 binder 引用建立一个 binder_ref_death 通知结构体，将其保存在当前进程的对应 binder 引用结构体的 death 域中。定义 binder_ref_death 的代码如下所示。

```
struct binder_ref_death {
        struct binder_work work;
        void __user *cookie;
};
```

5.1.5　结构体 binder_buffer

结构体 binder_buffer 用来描述一个内核缓冲区，能够在进程之间传输数据。定义 binder_buffer 的代码如下所示。

```
struct binder_buffer {
        struct list_head entry; /* free and allocated entries by address */
        struct rb_node rb_node; /* free entry by size or allocated entry */
                                /* by address */
        unsigned free:1;
        unsigned allow_user_free:1;
        unsigned async_transaction:1;
```



```
        unsigned debug_id:29;

        struct binder_transaction *transaction;

        struct binder_node *target_node;
        size_t data_size;
        size_t offsets_size;
        uint8_t data[0];
};
```

结构体 binder_buffer 能够存储 Binder 的相关信息，成员的具体说明如下所示。

- ☑ entry：构建一个双向链表。
- ☑ rb_node：表示一个红黑树节点。
- ☑ transaction：用于中转请求和返回结果。
- ☑ target_node：是一个目标节点。
- ☑ data_size：表示数据的大小。
- ☑ offsets_size：是一个偏移量。
- ☑ data[0]：用于存储实际数据。

5.1.6　结构体 binder_proc

结构体 binder_proc 表示正在使用 Binder 进程通信机制的进程，能够保存调用 Binder 的各个进程或线程的信息，例如线程 ID、进程 ID、Binder 状态信息等。定义 binder_proc 的具体实现代码如下所示。

```
struct binder_proc {
    //实现双向链表
    struct hlist_node proc_node;
    //线程队列、双向链表、所有的线程信息
    struct rb_root threads;
    struct rb_root nodes;
    struct rb_root refs_by_desc;
    struct rb_root refs_by_node;
    //进程 ID
    int pid;
    struct vm_area_struct *vma;
    struct task_struct *tsk;
    struct files_struct *files;
    struct hlist_node deferred_work_node;
    int deferred_work;
    void *buffer;
    ptrdiff_t user_buffer_offset;

    struct list_head buffers;
    struct rb_root free_buffers;
    struct rb_root allocated_buffers;
    size_t free_async_space;
```

```
	struct page **pages;
	size_t buffer_size;
	uint32_t buffer_free;
	struct list_head todo;
	//等待队列
	wait_queue_head_t wait;
	//Binder 状态
	struct binder_stats stats;
	struct list_head delivered_death;
	//最大线程
	int max_threads;
	int requested_threads;
	int requested_threads_started;
	int ready_threads;
	//默认优先级
	long default_priority;
};
```

在上述代码中，成员 proc_node 用于实现双向链表，成员 threads 用于存储所有的线程信息。

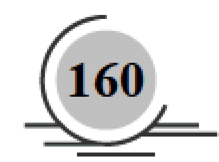
5.1.7　结构体 binder_thread

结构体 binder_thread 用于存储每一个单独线程的信息，表示 Binder 线程池中的一个线程。定义 binder_thread 的具体实现代码如下所示。

```
struct binder_thread {
		struct binder_proc *proc;
		struct rb_node rb_node;
		int pid;
		int looper;
		struct binder_transaction *transaction_stack;
		struct list_head todo;
		uint32_t return_error;
		uint32_t return_error2;
		wait_queue_head_t wait;
		struct binder_stats stats;
};
```

各个成员的具体说明如下所示。

☑ proc：表示当前线程属于哪一个 Binder 进程（binder_proc 指针）。
☑ rb_node：是一个红黑树节点。
☑ pid：表示线程的 pid。
☑ looper：表示线程的状态信息。
☑ transaction_stack：定义了要接收和发送的进程和线程信息，其结构体为 binder_transaction。
☑ todo：用于创建一个双向链表。
☑ return_error 和 return_error2：表示返回的错误信息代码。
☑ wait：是一个等待队列头结构，具体的定义代码如下所示。



```
struct binder_stats {
	int br[_IOC_NR(BR_FAILED_REPLY) + 1];
	int bc[_IOC_NR(BC_DEAD_BINDER_DONE) + 1];
	int obj_created[BINDER_STAT_COUNT];
	int obj_deleted[BINDER_STAT_COUNT];
};
```

各个成员的具体说明如下所示。

☑ br：用来存储 BINDER_WRITE_READ 的写操作命令协议（Binder Driver Return Protocol）。

☑ bc：存储着 BINDER_WRITE_READ 的写操作命令协议（Binder Driver Command Protocol）。

☑ obj_created：保存 BINDER_STAT_COUNT 的对象计数，当创建一个对象时需要同时调用该成员来增加相应的对象计数，而 obj_deleted 则正好与之相反。

looper 表示的线程状态信息在如下枚举中定义。

```
enum {
	BINDER_LOOPER_STATE_REGISTERED = 0x01,
	BINDER_LOOPER_STATE_ENTERED = 0x02,
	BINDER_LOOPER_STATE_EXITED = 0x04,
	BINDER_LOOPER_STATE_INVALID = 0x08,
	BINDER_LOOPER_STATE_WAITING = 0x10,
	BINDER_LOOPER_STATE_NEED_RETURN = 0x20
};
```

上述枚举主要包括的状态信息有注册、进入、退出、销毁、等待和需要返回。

5.1.8　结构体 binder_transaction

结构体 binder_transaction 的功能是中转请求和返回结果，并保存接收和要发送的进程信息。定义结构体 binder_transaction 的具体实现代码如下所示。

```
struct binder_transaction {
	int debug_id;//调试相关
	struct binder_work work;
	struct binder_thread *from;
	struct binder_transaction *from_parent;
	struct binder_proc *to_proc;
	struct binder_thread *to_thread;
	struct binder_transaction *to_parent;
	unsigned need_reply : 1;
	struct binder_buffer *buffer;
	unsigned int code;
	unsigned int flags;
	long priority;
	long saved_priority;
	uid_t sender_euid;
};
```

上述成员的具体说明如下所示。

☑ work：是一个 binder_work。

- ☑ from 和 to_thread：都是一个 binder_thread 对象，用于表示接收和要发送的进程信息。
- ☑ from_parent 和 to_thread：接收和发送进程信息的父节点。
- ☑ to_proc：是一个 binder_proc 类型的结构体，还包括 flags、need_reply、优先级（priority）等数据。
- ☑ sender_euid：Linux 系统中的每个进程都有两个 ID：用户 ID 和有效用户 ID，UID 一般表示进程的创建者（属于哪个用户创建），EUID 表示进程对于文件和资源的访问权限。sender_euid 表示要发送进程对文件和资源的操作权限。

另外在结构体 binder_transaction 中，还包含了类型类 inder_buffer 的一个 buffer，用来表示 binder 的缓冲区信息。inder_buffer 在前面已经进行了讲解。

5.1.9 结构体 binder_write_read

结构体 binder_write_read 的功能是表示在进程之间的通信过程中传输的数据，数据包中有一个 cmd 域用于区分不同的请求。定义结构体 binder_write_read 的实现代码如下所示。

```
struct binder_write_read {
        signed long write_size;       /* bytes to write */
        signed long write_consumed; /* bytes consumed by driver */
        unsigned long write_buffer;
        signed long read_size;        /* bytes to read */
        signed long read_consumed;    /* bytes consumed by driver */
        unsigned long read_buffer;
};
```

各个成员的具体说明如下所示。

- ☑ write_size 和 read_size：分别表示写入和读取的数据的大小。
- ☑ write_consumed 和 read_consumed：分别表示被消耗的写数据和读数据的大小。

当 Binder 驱动找到处理此事件的进程之后，Binder 驱动就会把需要处理的事件的任务放在读缓冲（binder_write_read）中，返回给这个服务线程，该服务线程则会执行指定命令的操作；处理请求的线程把数据交给合适的对象来执行预定操作，然后把返回结果同样用结构 binder_transaction_data 进行封装，以写命令的方式传回给 Binder 驱动，并将此数据放在一个读缓冲（binder_write_read）中，返回给正在等待结果的原进程（线程），这样就完成了一次通信。

5.1.10 BinderDriverCommandProtocol

结构体 binder_write_read 包含的命令在 BinderDriverCommandProtocol 中定义，具体代码如下所示。

```
enum BinderDriverCommandProtocol {
        BC_TRANSACTION = _IOW('c', 0, struct binder_transaction_data),
        BC_REPLY = _IOW('c', 1, struct binder_transaction_data),
        BC_ACQUIRE_RESULT = _IOW('c', 2, int),
        BC_FREE_BUFFER = _IOW('c', 3, int),
        BC_INCREFS = _IOW('c', 4, int),
        BC_ACQUIRE = _IOW('c', 5, int),
```

```
        BC_RELEASE = _IOW('c', 6, int),
        BC_DECREFS = _IOW('c', 7, int),
        BC_INCREFS_DONE = _IOW('c', 8, struct binder_ptr_cookie),
        BC_ACQUIRE_DONE = _IOW('c', 9, struct binder_ptr_cookie),
        BC_ATTEMPT_ACQUIRE = _IOW('c', 10, struct binder_pri_desc),
        BC_REGISTER_LOOPER = _IO('c', 11),
        BC_ENTER_LOOPER = _IO('c', 12),
        BC_EXIT_LOOPER = _IO('c', 13),
        BC_REQUEST_DEATH_NOTIFICATION = _IOW('c', 14, struct binder_ptr_cookie),
        BC_CLEAR_DEATH_NOTIFICATION = _IOW('c', 15, struct binder_ptr_cookie),
        BC_DEAD_BINDER_DONE = _IOW('c', 16, void *),
};
```

在上述枚举命令成员中，重要的是 BC_TRANSACTION 和 BC_REPLY 命令，被作为发送操作的命令，其数据参数都是 binder_transaction_data 结构体。其中前者用于翻译和解析将要被处理的事件数据，而后者则是事件处理完成之后对返回“结果数据”的操作命令。

5.1.11　枚举 BinderDriverReturnProtocol

在枚举 BinderDriverReturnProtocol 中定义了读操作命令协议，具体实现代码如下所示。

```
enum BinderDriverReturnProtocol {
        BR_ERROR = _IOR('r', 0, int),
        BR_OK = _IO('r', 1),
        /* No parameters! */
        BR_TRANSACTION = _IOR('r', 2, struct binder_transaction_data),
        BR_REPLY = _IOR('r', 3, struct binder_transaction_data),
        BR_ACQUIRE_RESULT = _IOR('r', 4, int),
        BR_DEAD_REPLY = _IO('r', 5),
        BR_TRANSACTION_COMPLETE = _IO('r', 6),
        BR_INCREFS = _IOR('r', 7, struct binder_ptr_cookie),
        BR_ACQUIRE = _IOR('r', 8, struct binder_ptr_cookie),
        BR_RELEASE = _IOR('r', 9, struct binder_ptr_cookie),
        BR_DECREFS = _IOR('r', 10, struct binder_ptr_cookie),
        BR_ATTEMPT_ACQUIRE = _IOR('r', 11, struct binder_pri_ptr_cookie),
        BR_NOOP = _IO('r', 12),
        BR_SPAWN_LOOPER = _IO('r', 13),
        BR_FINISHED = _IO('r', 14),
        BR_DEAD_BINDER = _IOR('r', 15, void *),
        BR_CLEAR_DEATH_NOTIFICATION_DONE = _IOR('r', 16, void *),
        BR_FAILED_REPLY = _IO('r', 17),
};
```

在上述命令中，BC_TRANSACTION 和 BC_REPLY 命令被作为发送操作命令，其数据参数都是 binder_transaction_data 结构体。其中，前者用于翻译和解析将要被处理的事件数据，而后者则是事件处理完成之后对返回“结果数据”的操作命令。

5.1.12 结构体 binder_ptr_cookie 和 binder_transaction_data

binder_ptr_cookie 和 binder_transaction_data 是两个比较重要的结构体，其中，binder_ptr_cookie 表示一个 Binder 实体对象或 Service 组件的死亡接收通知，具体定义代码如下所示。

```
struct binder_ptr_cookie {
	void *ptr;
	void *cookie;
};
```

而 binder_transaction_data 表示在通信过程中传递的数据，具体定义代码如下所示。

```
struct binder_transaction_data {
	union {
		size_t	handle; /* target descriptor of command transaction */
		void *ptr;	/* target descriptor of return transaction */
	} target;
	void *cookie;				/* target object cookie */
	unsigned int code;			/* transaction command */
	/* General information about the transaction. */
	unsigned int flags;
	pid_t sender_pid;
	uid_t sender_euid;
	size_t data_size;		/* number of bytes of data */
	size_t offsets_size;	/* number of bytes of offsets */
	union {
		struct {
			/* transaction data */
			const void *buffer;
			/* offsets from buffer to flat_binder_object structs */
			const void *offsets;
		} ptr;
		uint8_t buf[8];
	} data;
};
```

5.1.13 结构体 flat_binder_object

在 Android 系统中，将在进程之间传递的数据称为 Binder 对象，即 Binder Object。Binder 对象在对应源码中使用结构体 flat_binder_object 来表示，具体代码如下所示。

```
struct flat_binder_object {
	/* 8 bytes for large_flat_header */
	unsigned long type;
	unsigned long flags;
	/* 8 bytes of data. */
	union {
		void *binder;		/* local object */
		signed long handle;		/* remote object */
```

```
	};
	/* extra data associated with local object */
	void *cookie;
};
```

各个成员的具体说明如下所示。

- ☑ type：描述了 Binder 的类型，传输的数据是一个复用数据联合体。对于 Binder 类型来说，数据是一个 Binder 本地对象。
- ☑ handle：是一个远程的 handle 句柄。假如 A 有一个对象 O，对于 A 来说，O 就是一个本地的 Binder 对象；如果 B 想访问 A 的 O 对象，对于 B 来说，O 就是一个 handle。所以 handle 和 Binder 都指向 O。
- ☑ cookie：如果是本地对象，Binder 还可以带有额外的数据，这些数据将被保存到 cookie 字段中。
- ☑ flags：表示传输方式，例如同步和异步等，其值同样使用一个 enum 来表示，具体定义代码如下所示。

```
enum transaction_flags {
	TF_ONE_WAY = 0x01, /* this is a one-way call: async, no return */
	TF_ROOT_OBJECT = 0x04, /* contents are the component's root object */
	TF_STATUS_CODE = 0x08, /* contents are a 32-bit status code */
	TF_ACCEPT_FDS = 0x10, /* allow replies with file descriptors */
};
```

5.1.14　设备初始化

可以在文件 binder.c 中找到该初始化函数 binder_init()，具体定义代码如下所示。

```
static int __init binder_init(void)
{
	int ret;

	binder_deferred_workqueue = create_singlethread_workqueue("binder");
	if (!binder_deferred_workqueue)
		return -ENOMEM;

	binder_debugfs_dir_entry_root = debugfs_create_dir("binder", NULL);
	if (binder_debugfs_dir_entry_root)
		binder_debugfs_dir_entry_proc = debugfs_create_dir("proc",
						binder_debugfs_dir_entry_root);
	ret = misc_register(&binder_miscdev);
	if (binder_debugfs_dir_entry_root) {
		debugfs_create_file("state",
				S_IRUGO,
				binder_debugfs_dir_entry_root,
				NULL,
				&binder_state_fops);
		debugfs_create_file("stats",
				S_IRUGO,
				binder_debugfs_dir_entry_root,
```

```
                                                NULL,
                                                &binder_stats_fops);
                debugfs_create_file("transactions",
                                                S_IRUGO,
                                                binder_debugfs_dir_entry_root,
                                                NULL,
                                                &binder_transactions_fops);
                debugfs_create_file("transaction_log",
                                                S_IRUGO,
                                                binder_debugfs_dir_entry_root,
                                                &binder_transaction_log,
                                                &binder_transaction_log_fops);
                debugfs_create_file("failed_transaction_log",
                                                S_IRUGO,
                                                binder_debugfs_dir_entry_root,
                                                &binder_transaction_log_failed,
                                                &binder_transaction_log_fops);
        }
        return ret;
}
```

binder_init()是 Binder 驱动的初始化函数，在实现时需要调用设备驱动接口。Android Binder 设备驱动接口函数是 device_initcall()，使用 module_init 和 module_exit 是为了同时兼容支持静态编译的驱动模块（buildin）和动态编译的驱动模块（module）。Binder 使用 device_initcall 的目的就是不让 Binder 驱动支持动态编译，需要在内核（Kernel）做镜像。initcall 用于注册进行初始化的函数，如果的确需要将 Binder 驱动修改为动态的内核模块，可以直接将 device_initcall 修改为 module_init，并增加 module_exit 的驱动卸载接口函数。

在注册 Binder 驱动为 Misc 设备时，指定了 Binder 驱动的 miscdevice，具体实现代码如下所示。

```
static struct miscdevice binder_miscdev = {
        .minor = MISC_DYNAMIC_MINOR,
        .name = "binder",
        .fops = &binder_fops
};
```

Binder 设备的主设备号为 10，此设备号是动态获得的，各个参数的具体说明如下所示。

☑ MISC_DYNAMIC_MINOR：.minor 被设置为动态获得设备号 MISC_DYNAMIC_MINOR。
☑ .name：表示设备名称。
☑ file_operations：指定了该设备的 file_operations 结构体，定义代码如下所示。

```
static struct file_operations binder_fops = {
    .owner = THIS_MODULE,
    .poll = binder_poll,
    .unlocked_ioctl = binder_ioctl,
    .mmap = binder_mmap,
    .open = binder_open,
    .flush = binder_flush,
    .release = binder_release,
};
```

在 Android 系统中，任何驱动程序都具备向用户空间的程序提供操作接口的功能。这个接口是一个标准接口，Android Binder 驱动提供了操作设备文件（/dev/binder）的接口。正如 binder_fops 所描述的 file_operations 结构体一样，其中主要包括了 binder_poll、binder_ioctl、binder_mmap、binder_open、binder_flush 和 binder_release 等标准操作接口。

5.1.15 打开 Binder 设备文件

在 Android 系统的 Binder 机制中，函数 binder_open()的功能是打开 Binder 设备文件/dev/binder。在 Android 系统中，底层驱动的任何一个进程及线程都可以打开一个 Binder 设备，其打开过程的实现代码如下所示。

```
static int binder_open(struct inode *nodp, struct file *filp)
{
	struct binder_proc *proc;
	binder_debug(BINDER_DEBUG_OPEN_CLOSE, "binder_open: %d:%d\n",
			current->group_leader->pid, current->pid);
	proc = kzalloc(sizeof(*proc), GFP_KERNEL);
	if (proc == NULL)
		return -ENOMEM;
	get_task_struct(current);
	proc->tsk = current;
	INIT_LIST_HEAD(&proc->todo);
	init_waitqueue_head(&proc->wait);
	proc->default_priority = task_nice(current);
	binder_lock(__func__);
	binder_stats_created(BINDER_STAT_PROC);
	hlist_add_head(&proc->proc_node, &binder_procs);
	proc->pid = current->group_leader->pid;
	INIT_LIST_HEAD(&proc->delivered_death);
	filp->private_data = proc;
	binder_unlock(__func__);
	if (binder_debugfs_dir_entry_proc) {
		char strbuf[11];
		snprintf(strbuf, sizeof(strbuf), "%u", proc->pid);
		proc->debugfs_entry = debugfs_create_file(strbuf, S_IRUGO,
			binder_debugfs_dir_entry_proc, proc, &binder_proc_fops);
	}
	return 0;
}
```

函数 binder_open()的具体实现流程如下所示。

（1）创建并分配一个 binder_proc 空间来保存 Binder 数据。

（2）增加当前线程/进程的引用计数，给 binder_proc 的 tsk 字段赋值。

（3）实现 binder_proc 队列的初始化，主要包括：

☑ 使用 INIT_LIST_HEAD 初始化链表头 todo。

☑ 使用 init_waitqueue_head 初始化等待队列 wait。

☑ 设置默认优先级（default_priority）为当前进程的 nice 值（通过 task_nice 得到当前进程的 nice 值）。

（4）增加 BINDER_STAT_PROC 的对象计数，并通过 hlist_add_head 把创建的 binder_proc 对象添加到全局的 binder_proc 哈希表中，这样任何一个进程就都可以访问到其他进程的 binder_proc 对象。

（5）把当前进程（或线程）的线程组的 pid（pid 指向线程 id）赋值给 proc 的 pid 字段，同时把创建的 binder_proc 对象指针赋值给 filp 的 private_data 对象并保存起来。

（6）在 binder proc 目录中创建只读文件/proc/binder/proc/$pid，功能是输出当前 binder proc 对象的状态。文件名以 pid 命名，但是该 pid 字段并不是当前进程/线程的 id，而是线程组的 pid，表示是线程组中第一个线程的 pid（因为上面是将 current->group_leader->pid 赋值给该 pid 字段的），并且在创建该文件时也指定了操作该文件的函数接口为 binder_read_proc_proc，此函数的参数表示创建的 binder_proc 对象 proc。

再看函数 binder_release()，此函数与函数 binder_open()的功能相反。当 Binder 驱动退出时，通过函数 binder_release()来释放在打开以及其他操作过程中分配的空间，并且同时清理相关的数据信息。函数 binder_release()的具体实现代码如下所示。

```
static int binder_release(struct inode *nodp, struct file *filp)
{
		struct binder_proc *proc = filp->private_data;
		debugfs_remove(proc->debugfs_entry);
		binder_defer_work(proc, BINDER_DEFERRED_RELEASE);
		return 0;
}
```

在上述代码中，首先获取使用 private_data()数据的权限，找到当前进程、线程的 pid，这样可以得到在 open 过程中创建的以 pid 命名的用来输出当前 binder proc 对象的状态的只读文件；然后调用函数 remove_proc_entry()实现删除操作；最后通过函数 binder_defer_work()和其参数 BINDER_DEFERRED_RELEASE 释放整个 binder_proc 对象的数据和分配的空间。

5.1.16 实现内存映射

在 Android 系统中，当打开 Binder 设备文件/dev/binder 后，需要调用函数 mmap()把设备内存映射到用户进程地址空间中，这样就可以像操作用户内存那样操作设备内存。在 Binder 设备中，对内存的映射操作是有限制的，例如 Binder 不能映射具有写权限的内存区域，最大能映射 4MB 的内存区域等。在 Android 系统中，大多数设备本身具有设备映射的设备内存，或者是在驱动初始化时由 vmalloc 或 kmalloc 等内核内存函数分配的，在 mmap 操作时分配 Binder 的设备内存。

函数 mmap()实现分配功能的实现流程如下所示。

（1）在内核虚拟映射表上获取一个可以使用的区域。

（2）分配物理页，并把物理页映射到获取的虚拟空间上。

（3）每个进程/线程只能执行一次映射操作，后面的操作都会返回错误。

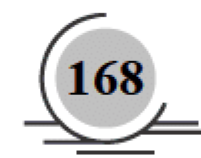
函数 mmap()的具体实现流程如下所示。

（1）检查内存映射条件，包括映射内存大小（4MB）、flags、是否是第一次 mmap 等。

（2）获得地址空间，并把此空间的地址记录在进程信息（buffer）中。

（3）分配物理页面（pages）并记录下来。

（4）将 buffer 插入到进程信息的 buffer 列表中。



（5）调用函数 binder_update_page_range()将分配的物理页面和 vm 空间对应起来。

（6）调用函数 binder_insert_free_buffer()把进程中的 buffer 插入到进程信息中。

函数 mmap()的具体实现代码如下所示。

```
static int binder_mmap(struct file *filp, struct vm_area_struct *vma)
{
	int ret;
	struct vm_struct *area;
	struct binder_proc *proc = filp->private_data;
	const char *failure_string;
	struct binder_buffer *buffer;

	if ((vma->vm_end - vma->vm_start) > SZ_4M)
		vma->vm_end = vma->vm_start + SZ_4M;

	binder_debug(BINDER_DEBUG_OPEN_CLOSE,
			"binder_mmap: %d %lx-%lx (%ld K) vma %lx pagep %lx\n",
			proc->pid, vma->vm_start, vma->vm_end,
			(vma->vm_end - vma->vm_start) / SZ_1K, vma->vm_flags,
			(unsigned long)pgprot_val(vma->vm_page_prot));

	if (vma->vm_flags & FORBIDDEN_MMAP_FLAGS) {
		ret = -EPERM;
		failure_string = "bad vm_flags";
		goto err_bad_arg;
	}
	vma->vm_flags = (vma->vm_flags | VM_DONTCOPY) & ~VM_MAYWRITE;

	mutex_lock(&binder_mmap_lock);
	if (proc->buffer) {
		ret = -EBUSY;
		failure_string = "already mapped";
		goto err_already_mapped;
	}

	area = get_vm_area(vma->vm_end - vma->vm_start, VM_IOREMAP);
	if (area == NULL) {
		ret = -ENOMEM;
		failure_string = "get_vm_area";
		goto err_get_vm_area_failed;
	}
	proc->buffer = area->addr;
	proc->user_buffer_offset = vma->vm_start - (uintptr_t)proc->buffer;
	mutex_unlock(&binder_mmap_lock);

#ifdef CONFIG_CPU_CACHE_VIPT
	if (cache_is_vipt_aliasing()) {
		while (CACHE_COLOUR((vma->vm_start ^ (uint32_t)proc->buffer))) {
			printk(KERN_INFO "binder_mmap: %d %lx-%lx maps %p bad alignment\n", proc->
pid, vma->vm_start, vma->vm_end, proc->buffer);
```

```
                        vma->vm_start += PAGE_SIZE;
                }
        }
#endif
        proc->pages = kzalloc(sizeof(proc->pages[0]) * ((vma->vm_end - vma->vm_start) / PAGE_SIZE),
GFP_KERNEL);
        if (proc->pages == NULL) {
                ret = -ENOMEM;
                failure_string = "alloc page array";
                goto err_alloc_pages_failed;
        }
        proc->buffer_size = vma->vm_end - vma->vm_start;

        vma->vm_ops = &binder_vm_ops;
        vma->vm_private_data = proc;

        if (binder_update_page_range(proc, 1, proc->buffer, proc->buffer + PAGE_SIZE, vma)) {
                ret = -ENOMEM;
                failure_string = "alloc small buf";
                goto err_alloc_small_buf_failed;
        }
        buffer = proc->buffer;
        INIT_LIST_HEAD(&proc->buffers);
        list_add(&buffer->entry, &proc->buffers);
        buffer->free = 1;
        binder_insert_free_buffer(proc, buffer);
        proc->free_async_space = proc->buffer_size / 2;
        barrier();
        proc->files = get_files_struct(proc->tsk);
        proc->vma = vma;
        proc->vma_vm_mm = vma->vm_mm;

        /*printk(KERN_INFO "binder_mmap: %d %lx-%lx maps %p\n",
                 proc->pid, vma->vm_start, vma->vm_end, proc->buffer);*/
        return 0;

err_alloc_small_buf_failed:
        kfree(proc->pages);
        proc->pages = NULL;
err_alloc_pages_failed:
        mutex_lock(&binder_mmap_lock);
        vfree(proc->buffer);
        proc->buffer = NULL;
err_get_vm_area_failed:
err_already_mapped:
        mutex_unlock(&binder_mmap_lock);
err_bad_arg:
        printk(KERN_ERR "binder_mmap: %d %lx-%lx %s failed %d\n",
               proc->pid, vma->vm_start, vma->vm_end, failure_string, ret);
        return ret;
}
```

在上述代码中，参数 vm_area_struct 是一个结构体，在 mmap 的具体实现中会非常有用。为了优化查找方法，内核专门维护了 VMA 的链表和树形结构。在结构 vm_area_struct 中，很多成员函数都是用来维护这个树形结构的。VMA 的功能是管理进程地址空间中不同区域的数据结构。该函数首先对内存映射进行检查，主要包括映射内存的大小、flags 以及是否已经映射过了，并判断其映射条件是否合法；然后，通过内核函数 get_vm_area()从系统中申请可用的虚拟内存空间，在内核中申请并保留一块连续的内核虚拟内存空间区域。接着将 binder_proc 的用户地址偏移（即用户进程的 VMA 地址与 Binder 申请的 VMA 地址的偏差）存放到 proc->user_buffer_offset 中；再使用 kzalloc()函数根据请求映射的内存空间大小，分配 Binder 的核心数据结构 binder_proc 的 pages 成员，主要用来保存指向申请的物理页的指针；最后，为 VMA 指定了 vm_operations_struct 操作，并且将 vma->vm_private_data 指向了核心数据 proc。

到目前为止，就可以真正地开始分配物理内存（page）了。物理内存的分配工作是通过函数 binder_update_page_range()实现的，该函数主要完成如下工作。

- ☑ alloc_page：分配页面。
- ☑ map_vm_area：为分配的内存做映射关系。
- ☑ vm_insert_page：把分配的物理页插入到用户 VMA 区域。

函数 binder_update_page_range()的具体实现代码如下所示。

```
static int binder_update_page_range(struct binder_proc *proc, int allocate,
                                    void *start, void *end,
                                    struct vm_area_struct *vma)
{
	void *page_addr;
	unsigned long user_page_addr;
	struct vm_struct tmp_area;
	struct page **page;
	struct mm_struct *mm;

	binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
		     "binder: %d: %s pages %p-%p\n", proc->pid,
		     allocate ? "allocate" : "free", start, end);

	if (end <= start)
		return 0;

	trace_binder_update_page_range(proc, allocate, start, end);

	if (vma)
		mm = NULL;
	else
		mm = get_task_mm(proc->tsk);

	if (mm) {
		down_write(&mm->mmap_sem);
		vma = proc->vma;
		if (vma && mm != proc->vma_vm_mm) {
			pr_err("binder: %d: vma mm and task mm mismatch\n",
```

```
                                        proc->pid);
                        vma = NULL;
                }
        }

        if (allocate == 0)
                goto free_range;

        if (vma == NULL) {
                printk(KERN_ERR "binder: %d: binder_alloc_buf failed to "
                        "map pages in userspace, no vma\n", proc->pid);
                goto err_no_vma;
        }

        for (page_addr = start; page_addr < end; page_addr += PAGE_SIZE) {
                int ret;
                struct page **page_array_ptr;
                page = &proc->pages[(page_addr - proc->buffer) / PAGE_SIZE];

                BUG_ON(*page);
                *page = alloc_page(GFP_KERNEL | __GFP_HIGHMEM | __GFP_ZERO);
                if (*page == NULL) {
                        printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
                                "for page at %p\n", proc->pid, page_addr);
                        goto err_alloc_page_failed;
                }
                tmp_area.addr = page_addr;
                tmp_area.size = PAGE_SIZE + PAGE_SIZE /* guard page? */;
                page_array_ptr = page;
                ret = map_vm_area(&tmp_area, PAGE_KERNEL, &page_array_ptr);
                if (ret) {
                        printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
                                "to map page at %p in kernel\n",
                                proc->pid, page_addr);
                        goto err_map_kernel_failed;
                }
                user_page_addr =
                        (uintptr_t)page_addr + proc->user_buffer_offset;
                ret = vm_insert_page(vma, user_page_addr, page[0]);
                if (ret) {
                        printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
                                "to map page at %lx in userspace\n",
                                proc->pid, user_page_addr);
                        goto err_vm_insert_page_failed;
                }
                /* vm_insert_page does not seem to increment the refcount */
        }
        if (mm) {
                up_write(&mm->mmap_sem);
                mmput(mm);
        }
```

```
        return 0;

free_range:
        for (page_addr = end - PAGE_SIZE; page_addr >= start;
             page_addr -= PAGE_SIZE) {
                page = &proc->pages[(page_addr - proc->buffer) / PAGE_SIZE];
                if (vma)
                        zap_page_range(vma, (uintptr_t)page_addr +
                                proc->user_buffer_offset, PAGE_SIZE, NULL);
err_vm_insert_page_failed:
                unmap_kernel_range((unsigned long)page_addr, PAGE_SIZE);
err_map_kernel_failed:
                __free_page(*page);
                *page = NULL;
err_alloc_page_failed:
                ;
        }
err_no_vma:
        if (mm) {
                up_write(&mm->mmap_sem);
                mmput(mm);
        }
        return -ENOMEM;
}
```

其中，vm_operations_struct 只包括了一个打开操作和一个关闭操作，具体的定义代码如下所示。

```
static struct vm_operations_struct binder_vm_ops = {
        .open = binder_vma_open,
        .close = binder_vma_close,
};
```

5.1.17　释放物理页面

在 Android 系统的 Binder 机制中，函数 binder_insert_free_buffer()的功能是将进程中的 buffer 插入到进程信息中。也就是说，通过此函数能够将一个空闲内核缓冲区加入到进程中的空闲内核缓冲区的红黑树中。函数 binder_insert_free_buffer()的具体实现代码如下所示。

```
static void binder_insert_free_buffer(struct binder_proc *proc,
                                      struct binder_buffer *new_buffer)
{
        struct rb_node **p = &proc->free_buffers.rb_node;
        struct rb_node *parent = NULL;
        struct binder_buffer *buffer;
        size_t buffer_size;
        size_t new_buffer_size;
        BUG_ON(!new_buffer->free);
        new_buffer_size = binder_buffer_size(proc, new_buffer);
        binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                     "binder: %d: add free buffer, size %zd, "
```

```
                    "at %p\n", proc->pid, new_buffer_size, new_buffer);
        while (*p) {
                parent = *p;
                buffer = rb_entry(parent, struct binder_buffer, rb_node);
                BUG_ON(!buffer->free);
                buffer_size = binder_buffer_size(proc, buffer);
                if (new_buffer_size < buffer_size)
                        p = &parent->rb_left;
                else
                        p = &parent->rb_right;
        }
        rb_link_node(&new_buffer->rb_node, parent, p);
        rb_insert_color(&new_buffer->rb_node, &proc->free_buffers);
}
```

5.1.18 分配内核缓冲区

在 Android 系统中，Binder 在使用 buffer 时一次声明一个 proc（对应一个进程）的 buffer 总大小，然后分配一页并做好映射。当在使用时如果发现空间不足，会接着映射并把这个 buffer 拆成两个，并把剩余的继续放到 free_buffers 中。在 Binder 驱动程序中，函数*binder_alloc_buf()的功能是分配内核缓冲区，具体代码如下所示。

```
static struct binder_buffer *binder_alloc_buf(struct binder_proc *proc,
                                              size_t data_size,
                                              size_t offsets_size, int is_async)
{
        struct rb_node *n = proc->free_buffers.rb_node;
        struct binder_buffer *buffer;
        size_t buffer_size;
        struct rb_node *best_fit = NULL;
        void *has_page_addr;
        void *end_page_addr;
        size_t size;
        if (proc->vma == NULL) {
                printk(KERN_ERR "binder: %d: binder_alloc_buf, no vma\n",
                       proc->pid);
                return NULL;
        }
        size = ALIGN(data_size, sizeof(void *)) +
                ALIGN(offsets_size, sizeof(void *));
        if (size < data_size || size < offsets_size) {
                binder_user_error("binder: %d: got transaction with invalid "
                        "size %zd-%zd\n", proc->pid, data_size, offsets_size);
                return NULL;
        }
        if (is_async &&
            proc->free_async_space < size + sizeof(struct binder_buffer)) {
                binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                             "binder: %d: binder_alloc_buf size %zd"
```

```
                    "failed, no async space left\n", proc->pid, size);
            return NULL;
    }
    while (n) {
            buffer = rb_entry(n, struct binder_buffer, rb_node);
            BUG_ON(!buffer->free);
            buffer_size = binder_buffer_size(proc, buffer);
            if (size < buffer_size) {
                    best_fit = n;
                    n = n->rb_left;
            } else if (size > buffer_size)
                    n = n->rb_right;
            else {
                    best_fit = n;
                    break;
            }
    }
    if (best_fit == NULL) {
            printk(KERN_ERR "binder: %d: binder_alloc_buf size %zd failed, "
                    "no address space\n", proc->pid, size);
            return NULL;
    }
    if (n == NULL) {
            buffer = rb_entry(best_fit, struct binder_buffer, rb_node);
            buffer_size = binder_buffer_size(proc, buffer);
    }
    binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                "binder: %d: binder_alloc_buf size %zd got buff"
                "er %p size %zd\n", proc->pid, size, buffer, buffer_size);
    has_page_addr =
            (void *)(((uintptr_t)buffer->data + buffer_size) & PAGE_MASK);
    if (n == NULL) {
            if (size + sizeof(struct binder_buffer) + 4 >= buffer_size)
                    buffer_size = size; /* no room for other buffers */
            else
                    buffer_size = size + sizeof(struct binder_buffer);
    }
    end_page_addr =
            (void *)PAGE_ALIGN((uintptr_t)buffer->data + buffer_size);
    if (end_page_addr > has_page_addr)
            end_page_addr = has_page_addr;
    if (binder_update_page_range(proc, 1,
        (void *)PAGE_ALIGN((uintptr_t)buffer->data), end_page_addr, NULL))
            return NULL;
    rb_erase(best_fit, &proc->free_buffers);
    buffer->free = 0;
    binder_insert_allocated_buffer(proc, buffer);
    if (buffer_size != size) {
            struct binder_buffer *new_buffer = (void *)buffer->data + size;
            list_add(&new_buffer->entry, &buffer->entry);
            new_buffer->free = 1;
```

```
                binder_insert_free_buffer(proc, new_buffer);
        }
        binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                     "binder: %d: binder_alloc_buf size %zd got "
                     "%p\n", proc->pid, size, buffer);
        buffer->data_size = data_size;
        buffer->offsets_size = offsets_size;
        buffer->async_transaction = is_async;
        if (is_async) {
                proc->free_async_space -= size + sizeof(struct binder_buffer);
                binder_debug(BINDER_DEBUG_BUFFER_ALLOC_ASYNC,
                             "binder: %d: binder_alloc_buf size %zd "
                             "async free %zd\n", proc->pid, size,
                             proc->free_async_space);
        }
        return buffer;
}
```

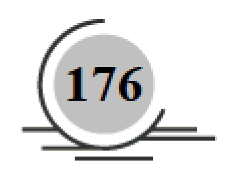
再看函数 binder_insert_allocated_buffer()，功能是将分配的内核缓冲区添加到目标进程的已分配物理页面的内核缓冲区红黑树中。函数 binder_insert_allocated_buffer()的具体实现代码如下所示。

```
static void binder_insert_allocated_buffer(struct binder_proc *proc,
                                           struct binder_buffer *new_buffer)
{
        struct rb_node **p = &proc->allocated_buffers.rb_node;
        struct rb_node *parent = NULL;
        struct binder_buffer *buffer;
        BUG_ON(new_buffer->free);
        while (*p) {
                parent = *p;
                buffer = rb_entry(parent, struct binder_buffer, rb_node);
                BUG_ON(buffer->free);
                if (new_buffer < buffer)
                        p = &parent->rb_left;
                else if (new_buffer > buffer)
                        p = &parent->rb_right;
                else
                        BUG();
        }
        rb_link_node(&new_buffer->rb_node, parent, p);
        rb_insert_color(&new_buffer->rb_node, &proc->allocated_buffers);
}
```

5.1.19 释放内核缓冲区

在 Android 系统中，函数 binder_free_buf()的功能是释放内核缓冲区，具体实现代码如下所示。

```
static void binder_free_buf(struct binder_proc *proc,
                            struct binder_buffer *buffer)
{
```



```
        size_t size, buffer_size;
        //计算要释放的内核缓冲区 buffer 的大小，保存在 buffer_size 中
        buffer_size = binder_buffer_size(proc, buffer);
        //计算数据缓冲区和偏移数组缓冲区的大小，并保存在 size 中
        size = ALIGN(buffer->data_size, sizeof(void *)) +
                ALIGN(buffer->offsets_size, sizeof(void *));

        binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                     "binder: %d: binder_free_buf %p size %zd buffer"
                     "_size %zd\n", proc->pid, buffer, size, buffer_size);

        BUG_ON(buffer->free);
        BUG_ON(size > buffer_size);
        BUG_ON(buffer->transaction != NULL);
        BUG_ON((void *)buffer < proc->buffer);
        BUG_ON((void *)buffer > proc->buffer + proc->buffer_size);
        //检查要释放的内核缓冲区 buffer 是否用于异步事务
        if (buffer->async_transaction) {
                proc->free_async_space += size + sizeof(struct binder_buffer);

                binder_debug(BINDER_DEBUG_BUFFER_ALLOC_ASYNC,
                             "binder: %d: binder_free_buf size %zd "
                             "async free %zd\n", proc->pid, size,
                             proc->free_async_space);
        }
        //释放内核缓冲区
        binder_update_page_range(proc, 0,
                (void *)PAGE_ALIGN((uintptr_t)buffer->data),
                (void *)(((uintptr_t)buffer->data + buffer_size) & PAGE_MASK),
                NULL);
        rb_erase(&buffer->rb_node, &proc->allocated_buffers);
        buffer->free = 1;
        if (!list_is_last(&buffer->entry, &proc->buffers)) {
                struct binder_buffer *next = list_entry(buffer->entry.next,
                                                struct binder_buffer, entry);
                if (next->free) {
                        rb_erase(&next->rb_node, &proc->free_buffers);
                        binder_delete_free_buffer(proc, next);
                }
        }
        if (proc->buffers.next != &buffer->entry) {
                struct binder_buffer *prev = list_entry(buffer->entry.prev,
                                                struct binder_buffer, entry);
                if (prev->free) {
                        binder_delete_free_buffer(proc, buffer);
                        rb_erase(&prev->rb_node, &proc->free_buffers);
                        buffer = prev;
                }
        }
        binder_insert_free_buffer(proc, buffer);
}
```

再看函数*buffer_start_page 和*buffer_end_page()，用于计算结构体 binder_buffer 所占用的虚拟页面的地址。具体实现代码如下所示。

```
static void *buffer_start_page(struct binder_buffer *buffer)
{
        return (void *)((uintptr_t)buffer & PAGE_MASK);
}
static void *buffer_end_page(struct binder_buffer *buffer)
{
        return (void *)(((uintptr_t)(buffer + 1) - 1) & PAGE_MASK);
}
```

再看函数 binder_delete_free_buffer()，功能是删除结构体 binder_buffer，具体实现代码如下所示。

```
static void binder_delete_free_buffer(struct binder_proc *proc,
                                      struct binder_buffer *buffer)
{
        struct binder_buffer *prev, *next = NULL;
        int free_page_end = 1;
        int free_page_start = 1;

        BUG_ON(proc->buffers.next == &buffer->entry);
        prev = list_entry(buffer->entry.prev, struct binder_buffer, entry);
        BUG_ON(!prev->free);
        if (buffer_end_page(prev) == buffer_start_page(buffer)) {
                free_page_start = 0;
                if (buffer_end_page(prev) == buffer_end_page(buffer))
                        free_page_end = 0;
                binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                             "binder: %d: merge free, buffer %p "
                             "share page with %p\n", proc->pid, buffer, prev);
        }

        if (!list_is_last(&buffer->entry, &proc->buffers)) {
                next = list_entry(buffer->entry.next,
                                  struct binder_buffer, entry);
                if (buffer_start_page(next) == buffer_end_page(buffer)) {
                        free_page_end = 0;
                        if (buffer_start_page(next) ==
                            buffer_start_page(buffer))
                                free_page_start = 0;
                        binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                                     "binder: %d: merge free, buffer"
                                     " %p share page with %p\n", proc->pid,
                                     buffer, prev);
                }
        }
        list_del(&buffer->entry);
        if (free_page_start || free_page_end) {
```

```
                binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                        "binder: %d: merge free, buffer %p do "
                        "not share page%s%s with with %p or %p\n",
                        proc->pid, buffer, free_page_start ? "" : " end",
                        free_page_end ? "" : " start", prev, next);
                binder_update_page_range(proc, 0, free_page_start ?
                    buffer_start_page(buffer) : buffer_end_page(buffer),
                    (free_page_end ? buffer_end_page(buffer) :
                    buffer_start_page(buffer)) + PAGE_SIZE, NULL);
        }
}
```

5.1.20　查询内核缓冲区

在 Android 系统中，函数*binder_buffer_lookup()的功能是根据一个用户空间地址查询一个内核缓冲区，具体实现代码如下所示。

```
static struct binder_buffer *binder_buffer_lookup(struct binder_proc *proc,
                                                  void __user *user_ptr)
{
        struct rb_node *n = proc->allocated_buffers.rb_node;
        //对于已经分配的 buffer 空间，以内存地址为索引加入红黑树 allocated_buffers 中
        struct binder_buffer *buffer;
        struct binder_buffer *kern_ptr;

        kern_ptr = user_ptr - proc->user_buffer_offset
                - offsetof(struct binder_buffer, data);
/* 进程 ioctl 传下来的指针并不是 binder_buffer 的地址，而直接是 binder_buffer.data 的地址。user_buffer_offset
用户空间和内核空间，被映射区域起始地址之间的偏移*/
        while (n) {
                buffer = rb_entry(n, struct binder_buffer, rb_node);
                BUG_ON(buffer->free);

                if (kern_ptr < buffer)
                        n = n->rb_left;
                else if (kern_ptr > buffer)
                        n = n->rb_right;
                else
                        return buffer;
        }
        return NULL;
}
```

5.2　Binder 封装库

在 Android 5.0 系统中，在各个层次都有和 Binder 有关的实现。其中主要的 Binder 库由本地原生

代码实现。本节将详细讲解 Binder 封装库的基本知识。

5.2.1 Binder 的 3 层结构

在 Android 系统中，Java 和 C++层都定义有同样功能的供应用程序使用的 Binder 接口，它们实际上都是调用原生 Binder 库的实现。各个实现层次的具体说明如下所示。

（1）Binder 驱动部分

驱动部分位于 Binder 结构的最底层（即 Linux 内核层），有关这部分的分析已经在本章前面进行了介绍。这部分用于实现 Binder 的设备驱动，主要实现如下功能。

☑ 组织 Binder 的服务节点。

☑ 调用 Binder 相关的处理线程。

☑ 完成实际的 Binder 传输。

（2）Binder Adapter 层

Binder Adapter 层是对 Binder 驱动的封装，主要功能是操作 Binder 驱动。应用程序无须直接和 Binder 驱动程序关联，关联文件包括 IPCThreadState.cpp、ProcessState.cpp 和 Parcel.cpp 中的一些内容。

Binder 核心库是 Binder 框架的核心实现，主要包括 IBinder、Binder（服务器端）和 BpBinder（客户端）。

（3）顶层

顶层的 Binder 框架和具体的客户端/服务端都分别有 Java 和 C++两种实现方案，主要供应用程序使用，例如摄像头和多媒体，这部分通过调用 Binder 的核心库来实现。

在文件 frameworks/native/include/binder/IInterface.h 中，分别定义了类 IInterface、类模板 BnInterface 和 BpInterface。其中，类模板 BnInterface 和 BpInterface 用于实现 Service 组件和 Client 组件，具体定义代码如下所示。

```
template<typename INTERFACE>
class BnInterface : public INTERFACE, public BBinder
{
public:
    virtual sp<IInterface> queryLocalInterface(const String16& _descriptor);
    virtual const String16& getInterfaceDescriptor() const;

protected:
    virtual IBinder* onAsBinder();
};

// ----------------------------------------------------------------------

template<typename INTERFACE>
class BpInterface : public INTERFACE, public BpRefBase
{
public:
    BpInterface(const sp<IBinder>& remote);

protected:
```

```
    virtual IBinder* onAsBinder();
};
```

在使用这两个模板时，起到了双继承的作用。使用者定义一个接口 interface，然后使用 BnInterface 和 BpInterface 两个模板结合自己的接口，构建自己的 BnXXX 和 BpXXX 两个类。

5.2.2 类 BBinder

类模板 BnInterface 继承于类 BBinder，BBinder 是服务的载体，和 binder 驱动共同工作，保证客户的请求最终是对一个 Binder 对象（BBinder 类）的调用。从 Binder 驱动的角度，每一个服务就是一个 BBinder 类，Binder 驱动负责找出服务对应的 BBinder 类，然后把这个 BBinder 类返回给 IPCThreadState，IPCThreadState 调用 BBinder 的 transact()。BBinder 的 transact()又会调用 onTransact()。BBinder::onTransact()是虚函数，所以实际是调用 BnXXXService::onTransact()，这样就可在 BnXXXService::onTransact()中完成具体的服务函数的调用。整个 BnXXXService 的类关系图如图 5-1 所示。

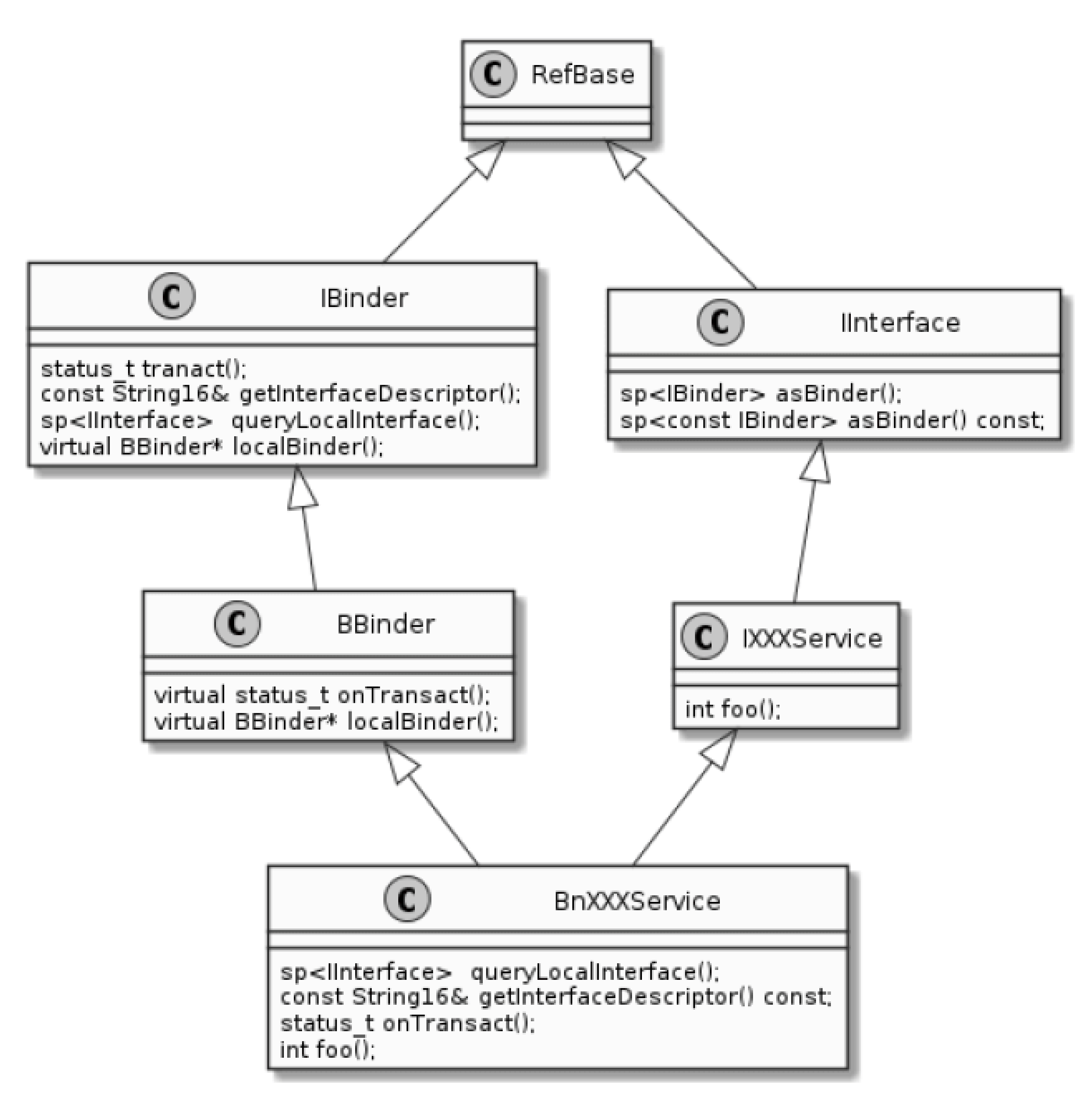


图 5-1 类关系图

由图 5-1 可以看出，BnXXXService 包含如下两部分。

☑ IXXXService：服务的主体的接口。

☑ BBinder：是服务的载体，和 Binder 驱动共同工作，保证客户的请求最终是对一个 Binder 对象（BBinder 类）的调用。

每一个服务就是一个 BBinder 类，Binder 驱动负责找出服务对应的 BBinder 类，然后把这个 BBinder 类返回给 IPCThreadState，IPCThreadState 调用 BBinder 的 transact()。BBinder 的 transact()又会调用 onTransact()。BBinder::onTransact()是虚函数，所以实际是调用 BnXXXService::onTransact()，这样就可在 BnXXXService::onTransact()中完成具体的服务函数的调用。

在文件 frameworks/native/include/binder/Binder.h 中，定义类 BBinder 的代码如下所示。

```
class BBinder : public IBinder
{
public:
                        BBinder();
    virtual const String16& getInterfaceDescriptor() const;
    virtual bool isBinderAlive() const;
    virtual status_t pingBinder();
    virtual status_t dump(int fd, const Vector<String16>& args);
    virtual status_t transact(    uint32_t code,
                                const Parcel& data,
                                Parcel* reply,
                                uint32_t flags = 0);
    virtual status_t linkToDeath(const sp<DeathRecipient>& recipient,
                                void* cookie = NULL,
                                uint32_t flags = 0);
    virtual status_t unlinkToDeath(    const wp<DeathRecipient>& recipient,
                                    void* cookie = NULL,
                                    uint32_t flags = 0,
                                    wp<DeathRecipient>* outRecipient = NULL);
    virtual void attachObject(    const void* objectID,
                                    void* object,
                                    void* cleanupCookie,
                                    object_cleanup_func func);
    virtual void* findObject(const void* objectID) const;
    virtual void detachObject(const void* objectID);
    virtual BBinder* localBinder();
protected:
    virtual ~BBinder();
    virtual status_t onTransact( uint32_t code,
                                const Parcel& data,
                                Parcel* reply,
                                uint32_t flags = 0);
private:
                        BBinder(const BBinder& o);
            BBinder& operator=(const BBinder& o);
    class Extras;
            Extras* mExtras;
            void* mReserved0;
};
```

在类 BBinder 中，当一个 Binder 代理对象通过 Binder 驱动程序向一个 Binder 本地对象发出一个进程通信请求时，Binder 驱动程序会调用该 Binder 本地对象的成员函数 transact()来处理这个请求。函数 transact()在文件 frameworks/native/libs/binder/Binder.cpp 中实现，具体代码如下所示。

```
status_t BBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    data.setDataPosition(0);
```

```
    status_t err = NO_ERROR;
    switch (code) {
        case PING_TRANSACTION:
            reply->writeInt32(pingBinder());
            break;
        default:
            err = onTransact(code, data, reply, flags);
            break;
    }
    if (reply != NULL) {
        reply->setDataPosition(0);
    }
    return err;
}
```

在上述代码中，PING_TRANSACTION 请求用来检查对象是否还存在，此处只是简单地把 pingBinder 的返回值返回给调用者，将其他的请求交给 onTransact 来处理。onTransact 是在 Bbinder 中声明的一个 protected 类型的虚函数，此功能在其子类中实现。

再看另外一个重要的成员函数 onTransact()，功能是分发和业务相关的进程间通信请求。函数 onTransact()也是在文件 frameworks/native/libs/binder/Binder.cpp 中定义的，具体实现代码如下所示。

```
status_t BBinder::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    switch (code) {
        case INTERFACE_TRANSACTION:
            reply->writeString16(getInterfaceDescriptor());
            return NO_ERROR;

        case DUMP_TRANSACTION: {
            int fd = data.readFileDescriptor();
            int argc = data.readInt32();
            Vector<String16> args;
            for (int i = 0; i < argc && data.dataAvail() > 0; i++) {
               args.add(data.readString16());
            }
            return dump(fd, args);
        }
        case SYSPROPS_TRANSACTION: {
            report_sysprop_change();
            return NO_ERROR;
        }
        default:
            return UNKNOWN_TRANSACTION;
    }
}
```

5.2.3　类 BpRefBase

类模板 BpInterface 继承于类 BpRefBase，起到代理作用。BpRefBase 以上是业务逻辑（要实现什

么功能)，BpRefBase 以下是数据传输(通过 binder 如何将功能实现)。在文件 frameworks/native/include/binder/Binder.h 中，定义类 BpRefBase 的代码如下所示。

```
class BpRefBase : public virtual RefBase
{
protected:
                                    BpRefBase(const sp<IBinder>& o);
    Virtual ~BpRefBase();
    virtual void onFirstRef();
    virtual void onLastStrongRef(const void* id);
    virtual bool onIncStrongAttempted(uint32_t flags, const void* id);

    inline    IBinder* remote(){ return mRemote; }
    inline    IBinder* remote() const { return mRemote; }

private:
                                    BpRefBase(const BpRefBase& o);
    BpRefBase& operator=(const BpRefBase& o);

    IBinder* const mRemote;
    RefBase::weakref_type* mRefs;
    volatile int32_t mState;
};

}; // namespace android
```

类 BpRefBase 中的成员函数 transact()用于向运行在 Server 进程中的 Service 组件发送进程之间的通信请求，这是通过 Binder 驱动程序间接实现的。函数 transact()的具体实现代码如下所示。

```
status_t BpBinder::transact(
uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
if (mAlive) {
status_t status = IPCThreadState::self()->transact(
mHandle, code, data, reply, flags);
if (status == DEAD_OBJECT) mAlive = 0;
return status;
}
return DEAD_OBJECT;
}
```

各个参数的具体说明如下所示。

☑ code：表示请求的 ID 号。

☑ data：表示请求的参数。

☑ reply：表示返回的结果。

☑ flags：是一些额外的标识，例如 FLAG_ONEWAY，通常为 0。

5.2.4　类 IPCThreadState

前面介绍的类 BBinder 和类 BpRefBase 都是通过类 IPCThreadState 和 Binder 的驱动程序交互实现的。类 IPCThreadState 在文件 frameworks/native/include/binder/IPCThreadState.h 中实现，具体实现代码如下所示。

```
class IPCThreadState
{
public:
    static    IPCThreadState* self();
    static    IPCThreadState* selfOrNull();    // self(), but won't instantiate

              sp<ProcessState> process();

              status_t clearLastError();

              int getCallingPid();
              int getCallingUid();

              void setStrictModePolicy(int32_t policy);
              int32_t getStrictModePolicy() const;

              void setLastTransactionBinderFlags(int32_t flags);
              int32_t getLastTransactionBinderFlags() const;

              int64_t clearCallingIdentity();
              void restoreCallingIdentity(int64_t token);

              void flushCommands();

              void joinThreadPool(bool isMain = true);

              void stopProcess(bool immediate = true);

              status_t transact(int32_t handle,
                                          uint32_t code, const Parcel& data,
                                          Parcel* reply, uint32_t flags);

              void incStrongHandle(int32_t handle);
              void decStrongHandle(int32_t handle);
              void incWeakHandle(int32_t handle);
              void decWeakHandle(int32_t handle);
              status_t attemptIncStrongHandle(int32_t handle);
    static    void expungeHandle(int32_t handle, IBinder* binder);
              status_t requestDeathNotification(     int32_t handle,
                                                          BpBinder* proxy);
              status_t clearDeathNotification( int32_t handle,
```

```
                                BpBinder* proxy);

    static    void shutdown();
    static    void disableBackgroundScheduling(bool disable);

private:
                        IPCThreadState();
                        ~IPCThreadState();

            status_t sendReply(const Parcel& reply, uint32_t flags);
            status_t waitForResponse(Parcel *reply,
                                    status_t *acquireResult=NULL);
            status_t talkWithDriver(bool doReceive=true);
            status_t writeTransactionData(int32_t cmd,
                                        uint32_t binderFlags,
                                        int32_t handle,
                                        uint32_t code,
                                        const Parcel& data,
                                        status_t* statusBuffer);
            status_t executeCommand(int32_t command);

            void clearCaller();

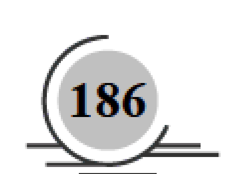
    static    void threadDestructor(void *st);
    static    void freeBuffer(Parcel* parcel,
                                const uint8_t* data, size_t dataSize,
                                const size_t* objects, size_t objectsSize,
                                void* cookie);

    const sp<ProcessState> mProcess;
    const pid_t mMyThreadId;
            Vector<BBinder*> mPendingStrongDerefs;
            Vector<RefBase::weakref_type*> mPendingWeakDerefs;

            Parcel mIn;
            Parcel mOut;
            status_t mLastError;
            pid_t mCallingPid;
            uid_t mCallingUid;
            int32_t mStrictModePolicy;
            int32_t mLastTransactionBinderFlags;
};

};
```

在类 IPCThreadState 中，成员函数用于实现数据处理。在 transact 请求中将请求的数据经过 Binder 设备发送给 Service，Service 处理完请求后，又将结果原路返回给客户端。函数 transact()在文件 frameworks/native/libs/binder/IPCThreadState.cpp 中定义，具体实现代码如下所示。



```
status_t IPCThreadState::transact(int32_t handle,
                                  uint32_t code, const Parcel& data,
                                  Parcel* reply, uint32_t flags)
{
    status_t err = data.errorCheck();
    flags |= TF_ACCEPT_FDS;
    IF_LOG_TRANSACTIONS() {
        TextOutput::Bundle _b(alog);
        alog << "BC_TRANSACTION thr " << (void*)pthread_self() << " / hand "
            << handle << " / code " << TypeCode(code) << ": "
            << indent << data << dedent << endl;
    }

    if (err == NO_ERROR) {
        LOG_ONEWAY(">>>> SEND from pid %d uid %d %s", getpid(), getuid(),
            (flags & TF_ONE_WAY) == 0 ? "READ REPLY" : "ONE WAY");
        err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);
    }

    if (err != NO_ERROR) {
        if (reply) reply->setError(err);
        return (mLastError = err);
    }

    if ((flags & TF_ONE_WAY) == 0) {
        #if 0
        if (code == 4) { // relayout
            ALOGI(">>>>>> CALLING transaction 4");
        } else {
            ALOGI(">>>>>> CALLING transaction %d", code);
        }
        #endif
        if (reply) {
            err = waitForResponse(reply);
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }
        #if 0
        if (code == 4) { // relayout
            ALOGI("<<<<<< RETURNING transaction 4");
        } else {
            ALOGI("<<<<<< RETURNING transaction %d", code);
        }
        #endif

        IF_LOG_TRANSACTIONS() {
            TextOutput::Bundle _b(alog);
            alog << "BR_REPLY thr " << (void*)pthread_self() << " / hand "
                << handle << ": ";
            if (reply) alog << indent << *reply << dedent << endl;
```

```
                else alog << "(none requested)" << endl;
            }
        } else {
            err = waitForResponse(NULL, NULL);
        }
        return err;
}
```

5.3 初始化 Java 层 Binder 框架

虽然 Java 层 Binder 系统是 Native 层 Binder 系统的一个镜像，但这个镜像终归还需借助 Native 层 Binder 系统来开展工作，即它和 Native 层 Binder 有着千丝万缕的关系，故一定要在 Java 层 Binder 正式工作之前建立这种关系。本节将详细讲解 Java 层 Binder 框架的初始化过程。

5.3.1 搭建交互关系

在 Android 系统中，函数 register_android_os_Binder()专门负责搭建 Java Binder 和 Native Binder 的交互关系。此函数在文件/frameworks/base/core/jni/android_util_Binder.cpp 中实现。

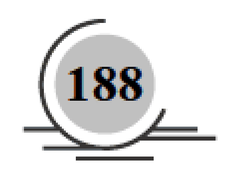
函数 register_android_os_Binder 的具体实现代码如下所示。

```
int register_android_os_Binder(JNIEnv* env)
{
   //初始化 Java Binder 类和 Native 层的关系
   if (int_register_android_os_Binder(env) < 0)
          return -1;
   //初始化 Java BinderInternal 类和 Native 层的关系
   if (int_register_android_os_BinderInternal(env) < 0)
          return -1;
   //初始化 Java BinderProxy 类和 Native 层的关系
   if (int_register_android_os_BinderProxy(env) < 0)
          return -1;
   //初始化 Java Parcel 类和 Native 层的关系
   if (int_register_android_os_Parcel(env) < 0)
          return -1;
   return 0;
}
```

根据上面的代码可知，函数 register_android_os_Binder()完成了 Java 层 Binder 架构中最重要的 4 个类的初始化工作。下面将详细分析 Binder 类的初始化进程。

5.3.2 实现 Binder 类的初始化

函数 int_register_android_os_Binder()实现了 Binder 类的初始化工作，此函数在文件 android_util_Binder.cpp 中实现，具体实现代码如下所示。



```
static int int_register_android_os_Binder(JNIEnv* env)
{
jclass clazz;
//kBinderPathName 为 Java 层中 Binder 类的全路径名，android/os/Binder
clazz = env->FindClass(kBinderPathName);
/*
gBinderOffsets 是一个静态类对象，专门保存 Binder 类的一些在 JNI 层中使用的信息，
如成员函数 execTransact()的 methodID，Binder 类中成员 mObject 的 fieldID

*/
  gBinderOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
  gBinderOffsets.mExecTransact
                    = env->GetMethodID(clazz, "execTransact", "(IIII)Z");
  gBinderOffsets.mObject
                    = env->GetFieldID(clazz, "mObject", "I");
  //注册 Binder 类中 native 函数的实现
  return AndroidRuntime::registerNativeMethods(
                            env, kBinderPathName,
                            gBinderMethods, NELEM(gBinderMethods));
}
```

从上面的代码可知，gBinderOffsets 对象保存了和 Binder 类相关的某些在 JNI 层中使用的信息。

下一个初始化的类是 BinderInternal，其代码位于 int_register_android_os_BinderInternal()函数中。此函数在文件 android_util_Binder.cpp 中实现，具体实现代码如下所示。

```
static int int_register_android_os_BinderInternal(JNIEnv* env)
{
 jclass clazz;
 //根据 BinderInternal 的全路径名找到代表该类的 jclass 对象。全路径名为
 // "com/android/internal/os/BinderInternal"
 clazz = env->FindClass(kBinderInternalPathName);
 //gBinderInternalOffsets 也是一个静态对象，用来保存 BinderInternal 类的一些信息
 gBinderInternalOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
 //获取 forceBinderGc 的 methodID
 gBinderInternalOffsets.mForceGc
               = env->GetStaticMethodID(clazz, "forceBinderGc", "()V");
  //注册 BinderInternal 类中 native 函数的实现
  return AndroidRuntime::registerNativeMethods(
                         env, kBinderInternalPathName,
                         gBinderInternalMethods, NELEM(gBinderInternalMethods));
}
```

由此可见，int_register_android_os_BinderInternal 的功能和 int_register_android_os_Binder 的功能类似，主要包括以下两个方面。

☑　获取一些有用的 methodID 和 fieldID。这表明 JNI 层一定会向上调用 Java 层的函数。

☑　注册相关类中 native 函数的实现。

5.3.3 实现 BinderProxy 类的初始化

函数 int_register_android_os_BinderProxy()完成了 BinderProxy 类的初始化工作，此函数在文件 android_util_Binder.cpp 中实现，具体实现代码如下所示。

```
static int int_register_android_os_BinderProxy(JNIEnv* env)
{
 jclass clazz;

 clazz = env->FindClass("java/lang/ref/WeakReference");
 //gWeakReferenceOffsets 用来和 WeakReference 类交互
 gWeakReferenceOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
 //获取 WeakReference 类 get()函数的 methodID
  gWeakReferenceOffsets.mGet= env->GetMethodID(clazz, "get",
                                          "()Ljava/lang/Object;");
  clazz = env->FindClass("java/lang/Error");
  //gErrorOffsets 用来和 Error 类交互
  gErrorOffsets.mClass = (jclass) env->NewGlobalRef(clazz);

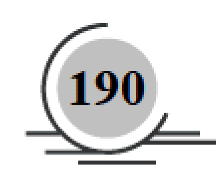
  clazz = env->FindClass(kBinderProxyPathName);
  //gBinderProxyOffsets 用来和 BinderProxy 类打交道
  gBinderProxyOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
  gBinderProxyOffsets.mConstructor= env->GetMethodID(clazz, "<init>", "()V");
  ...... //获取 BinderProxy 的一些信息
  clazz = env->FindClass("java/lang/Class");
  //gClassOffsets 用来和 Class 类交互
  gClassOffsets.mGetName =env->GetMethodID(clazz,
                                   "getName", "()Ljava/lang/String;");
  //注册 BinderProxy native 函数的实现
  return AndroidRuntime::registerNativeMethods(env,
           kBinderProxyPathName,gBinderProxyMethods,
                                      NELEM(gBinderProxyMethods));
}
```

根据上面的代码可知，int_register_android_os_BinderProxy()函数除了初始化 BinderProxy 类外，还获取了 WeakReference 类和 Error 类的一些信息。由此可见，BinderProxy 对象的生命周期会委托 WeakReference 来管理，故 JNI 层会获取该类 get()函数的 MethodID。

到此为止，Java Binder 几个重要成员的初始化已完成，同时在代码中定义了几个全局静态对象，分别是 gBinderOffsets、gBinderInternalOffsets 和 gBinderProxyOffsets。

5.4 实体对象 binder_node

在 Android 系统中，binder_node 用来描述一个 Binder 实体对象。每个 Service 组件在 Binder 驱动程序中都对应有一个 Binder 实体对象，用来描述它在内核中的状态。Android 系统的 Binder 通信框架如图 5-2 所示。



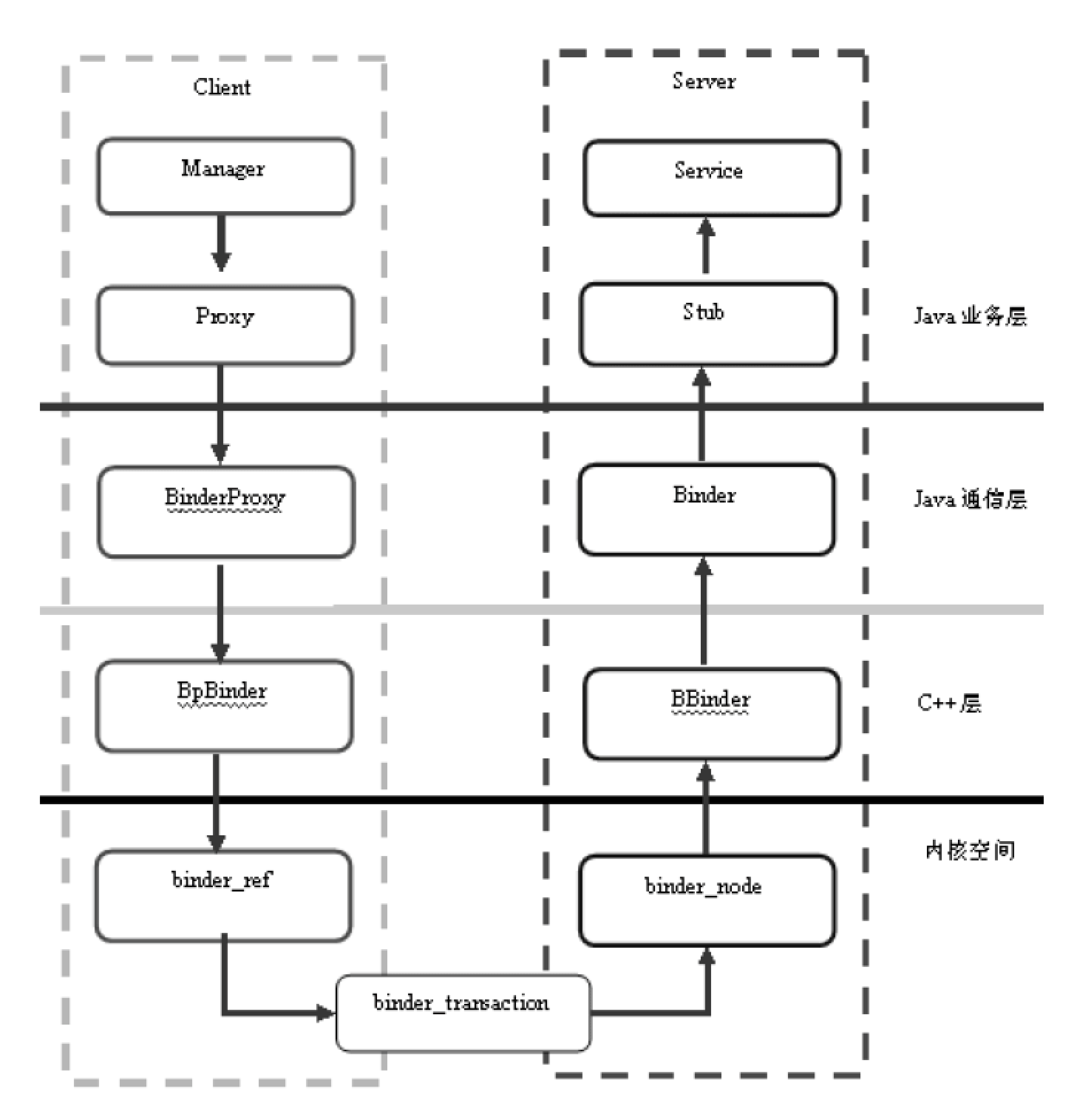


图 5-2　Binder 通信框架图

本节将详细讲解 Android 5.0 实体对象 binder_node 的知识。

5.4.1　定义实体对象

Binder 实体对象 binder_node 的定义代码如下所示。

```
struct binder_node {
	//调试 id
	int debug_id;
	//描述一个待处理的工作项
	struct binder_work work;
	union {
		//挂载到宿主进程 binder_proc 的成员变量 nodes 红黑树的节点
		struct rb_node rb_node;
		//当宿主进程死亡，该 Binder 实体对象将挂载到全局 binder_dead_nodes 链表中
		struct hlist_node dead_node;
	};
	//指向该 Binder 线程的宿主进程
	struct binder_proc *proc;
	//保存所有引用该 Binder 实体对象的 Binder 引用对象
	struct hlist_head refs;
	//Binder 实体对象的强引用计数
	int internal_strong_refs;
	int local_strong_refs;
```

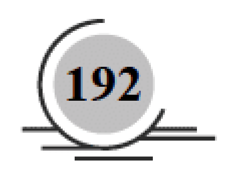
```
    unsigned has_strong_ref:1;
    unsigned pending_strong_ref:1;
    unsigned has_weak_ref:1;
    unsigned pending_weak_ref:1;
    //Binder 实体对象的弱引用计数
    int local_weak_refs;
    //指向用户空间 service 组件内部的引用计数对象 wekref_impl 的地址
    void __user *ptr;
    //保存用户空间的 service 组件地址
    void __user *cookie;
    //标示该 Binder 实体对象是否正在处理一个异步事务
    unsigned has_async_transaction:1;
    //设置该 Binder 实体对象是否可以接收包含有文件描述符的 IPC 数据
    unsigned accept_fds:1;
    //Binder 实体对象要求处理线程应具备的最小线程优先级
    unsigned min_priority:8;
    //异步事务队列
    struct list_head async_todo;
};
```

通过上述代码可知，在 Binder 驱动中，用户空间中的每一个 Binder 本地对象都对应有一个 Binder 实体对象。各个成员的具体说明如下所示。

- ☑ proc：指向 Binder 实体对象的宿主进程，宿主进程使用红黑树来维护其内部的所有 Binder 实体对象。
- ☑ rb_node：用来挂载到宿主进程 proc 的 Binder 实体对象红黑树中的节点。
- ☑ dead_node：如果该 Binder 实体对象的宿主进程已经死亡，该 Binder 实体就通过成员变量 dead_node 保存到全局链表 binder_dead_nodes。
- ☑ refs：一个 Binder 实体对象可以被多个 client 引用，成员变量 refs 用来保存所有引用该 Binder 实体的 Binder 引用对象。
- ☑ internal_strong_refs 和 local_strong_refs：都是用来描述 Binder 实体对象的强引用计数。
- ☑ local_weak_refs：用来描述 Binder 实体对象的弱引用计数。
- ☑ ptr 和 cookie：分别指向用户空间地址，cookie 指向 BBinder 的地址，ptr 指向 BBinder 对象的引用计数地址。
- ☑ has_async_transaction：描述一个 Binder 实体对象是否正在处理一个异步事务，当 Binder 驱动指定某个线程来处理某一事务时，首先将该事务保存到指定线程的 todo 队列中，表示要由该线程来处理该事务。如果是异步事务，Binder 驱动程序就会将它保存在目标 Binder 实体对象的一个异步事务队列 async_todo 中。
- ☑ min_priority：表示一个 Binder 实体对象在处理来自 client 进程请求时，要求处理线程的最小线程优先级。

5.4.2 增加引用计数

在 Binder 驱动程序中，使用函数 binder_inc_node()来增加一个 Binder 实体对象的引用计数。函数 binder_inc_node()在文件/drivers/staging/android/binder.c 中定义，具体实现代码如下所示。



```
static int binder_inc_node(struct binder_node *node, int strong, int internal,
                           struct list_head *target_list)
{
        if (strong) {
                if (internal) {
                        if (target_list == NULL &&
                            node->internal_strong_refs == 0 &&
                            !(node == binder_context_mgr_node &&
                            node->has_strong_ref)) {
                                printk(KERN_ERR "binder: invalid inc strong "
                                        "node for %d\n", node->debug_id);
                                return -EINVAL;
                        }
                        node->internal_strong_refs++;
                } else
                        node->local_strong_refs++;
                if (!node->has_strong_ref && target_list) {
                        list_del_init(&node->work.entry);
                        list_add_tail(&node->work.entry, target_list);
                }
        } else {
                if (!internal)
                        node->local_weak_refs++;
                if (!node->has_weak_ref && list_empty(&node->work.entry)) {
                        if (target_list == NULL) {
                                printk(KERN_ERR "binder: invalid inc weak node "
                                        "for %d\n", node->debug_id);
                                return -EINVAL;
                        }
                        list_add_tail(&node->work.entry, target_list);
                }
        }
        return 0;
}
```

各个参数的具体说明如下所示。

- ☑ node：表示要增加引用计数的 Binder 实体对象。
- ☑ strong：表示要增加强引用计数还是要增加弱引用计数。
- ☑ internal：用于区分增加的是内部引用计数还是外部引用计数。
- ☑ target_list：用于指向一个目标进程或目标线程的 todo 队列，当不是 null 值时，表示增加了 Binder 实体对象的引用计数后，需要对应增加它所引用的 Binder 本地对象的引用计数。

5.4.3　减少引用计数

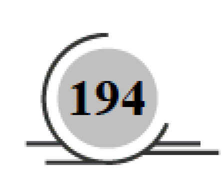
与函数 binder_inc_node()相反，在 Binder 驱动程序中，使用函数 binder_dec_node()来减少一个 Binder 实体对象的引用计数。函数 binder_dec_node()会减少 internal_strong_refs、local_strong_refs 或 local_weak_refs 的使用计数，并删除节点的 work.entry 链表。函数 binder_dec_node()也是在文件/drivers/staging/

android/binder.c 中定义，具体实现代码如下所示。

```
static int binder_dec_node(struct binder_node *node, int strong, int internal)
{
	if (strong) {
		if (internal)
			node->internal_strong_refs--;
		else
			node->local_strong_refs--;
		if (node->local_strong_refs || node->internal_strong_refs)
			return 0;
	} else {
		if (!internal)
			node->local_weak_refs--;
		if (node->local_weak_refs || !hlist_empty(&node->refs))
			return 0;
	}
	if (node->proc && (node->has_strong_ref || node->has_weak_ref)) {
		if (list_empty(&node->work.entry)) {
			list_add_tail(&node->work.entry, &node->proc->todo);
			wake_up_interruptible(&node->proc->wait);
		}
	} else {
		if (hlist_empty(&node->refs) && !node->local_strong_refs &&
		    !node->local_weak_refs) {
			list_del_init(&node->work.entry);
			if (node->proc) {
				rb_erase(&node->rb_node, &node->proc->nodes);
				binder_debug(BINDER_DEBUG_INTERNAL_REFS,
					     "binder: refless node %d deleted\n",
					     node->debug_id);
			} else {
				hlist_del(&node->dead_node);
				binder_debug(BINDER_DEBUG_INTERNAL_REFS,
					     "binder: dead node %d deleted\n",
					     node->debug_id);
			}
			kfree(node);
			binder_stats_deleted(BINDER_STAT_NODE);
		}
	}
	return 0;
}
```

5.5　本地对象 BBinder

因为 Binder 的功能就是在本地执行其他进程的功能，所以对于 Binder 机制来说，不但是 Android 系统中的一个完美的 IPC 机制，其实也是 Android 的一种远程过程调用（RPC）机制。当进程通过 Binder



获取将要调用的进程服务时，不但可以是一个本地对象，也可以是一个远程服务的引用。也就是说，Binder 不但可以与本地进程通信，还可以与远程进程通信。此处的本地进程就是本节所讲解的本地对象，而远程进程就是远程服务的一个引用。

5.5.1　引用了运行的本地对象

在 Android 系统中，Binder 驱动程序通过函数 binder_thread_read()引用了运行在 Server 进程中的 Binder 本地对象，此函数在文件 drivers/staging/android/binder.c 中定义。当 service_manager 运行时此函数会一直等待，直到有请求到达为止。函数 binder_thread_read()的具体实现代码如下所示。

```
static int binder_thread_read(struct binder_proc *proc,
                              struct binder_thread *thread,
                              void  __user *buffer, int size,
                              signed long *consumed, int non_block)
{
        void __user *ptr = buffer + *consumed;
        void __user *end = buffer + size;
        int ret = 0;
        int wait_for_proc_work;
        if (*consumed == 0) {
                if (put_user(BR_NOOP, (uint32_t __user *)ptr))
                        return -EFAULT;
                ptr += sizeof(uint32_t);
        }
retry:
        wait_for_proc_work = thread->transaction_stack == NULL &&
                                list_empty(&thread->todo);
        if (thread->return_error != BR_OK && ptr < end) {
                if (thread->return_error2 != BR_OK) {
                        if (put_user(thread->return_error2, (uint32_t __user *)ptr))
                                return -EFAULT;
                        ptr += sizeof(uint32_t);
                        binder_stat_br(proc, thread, thread->return_error2);
                        if (ptr == end)
                                goto done;
                        thread->return_error2 = BR_OK;
                }
                if (put_user(thread->return_error, (uint32_t __user *)ptr))
                        return -EFAULT;
                ptr += sizeof(uint32_t);
                binder_stat_br(proc, thread, thread->return_error);
                thread->return_error = BR_OK;
                goto done;
        }
        thread->looper |= BINDER_LOOPER_STATE_WAITING;
        if (wait_for_proc_work)
                proc->ready_threads++;
        binder_unlock(__func__);
        trace_binder_wait_for_work(wait_for_proc_work,
```

```
                                            !!thread->transaction_stack,
                                            !list_empty(&thread->todo));
        if (wait_for_proc_work) {
                if (!(thread->looper & (BINDER_LOOPER_STATE_REGISTERED |
                                              BINDER_LOOPER_STATE_ENTERED))) {
                        binder_user_error("binder: %d:%d ERROR: Thread waiting "
                                "for process work before calling BC_REGISTER_"
                                "LOOPER or BC_ENTER_LOOPER (state %x)\n",
                                proc->pid, thread->pid, thread->looper);
                        wait_event_interruptible(binder_user_error_wait,
                                                 binder_stop_on_user_error < 2);
                }
                binder_set_nice(proc->default_priority);
                if (non_block) {
                        if (!binder_has_proc_work(proc, thread))
                                ret = -EAGAIN;
                } else
                        ret = wait_event_freezable_exclusive(proc->wait, binder_has_proc_work(proc, thread));
        } else {
                if (non_block) {
                        if (!binder_has_thread_work(thread))
                                ret = -EAGAIN;
                } else
                        ret = wait_event_freezable(thread->wait, binder_has_thread_work(thread));
        }
        binder_lock(__func__);
        if (wait_for_proc_work)
                proc->ready_threads--;
        thread->looper &= ~BINDER_LOOPER_STATE_WAITING;
        if (ret)
                return ret;
        while (1) {
                uint32_t cmd;
//将用户传进来的 transact 参数复制到本地变量 struct binder_transaction_data tr 中
                struct binder_transaction_data tr;
                struct binder_work *w;
                struct binder_transaction *t = NULL;
//由于 thread->todo 不为空，执行下列语句
                if (!list_empty(&thread->todo))
                        w = list_first_entry(&thread->todo, struct binder_work, entry);
                else if (!list_empty(&proc->todo) && wait_for_proc_work)
//Service Manager 被唤醒之后，就进入 while 循环开始处理事务了
//此处 wait_for_proc_work 等于 1，并且 proc->todo 不为空，所以从 proc->todo 列表中得到第一个工作项
                        w = list_first_entry(&proc->todo, struct binder_work, entry);
                else {
                        if (ptr - buffer == 4 && !(thread->looper & BINDER_LOOPER_STATE_NEED_
RETURN)) /* no data added */
                                goto retry;
                        break;
                }
                if (end - ptr < sizeof(tr) + 4)
```

```
                    break;
            switch (w->type) {
//函数调用 binder_transaction 进一步处理
            case BINDER_WORK_TRANSACTION: {
//因为这个工作项的类型为 BINDER_WORK_TRANSACTION，所以通过下面语句得到事务项
                    t = container_of(w, struct binder_transaction, work);
            } break;
//因为 w->type 为 BINDER_WORK_TRANSACTION_COMPLETE
//这是在上面的 binder_transaction()函数中设置的，于是执行下面的代码
            case BINDER_WORK_TRANSACTION_COMPLETE: {
                    cmd = BR_TRANSACTION_COMPLETE;
                    if (put_user(cmd, (uint32_t __user *)ptr))
                            return -EFAULT;
                    ptr += sizeof(uint32_t);
                    binder_stat_br(proc, thread, cmd);
                    binder_debug(BINDER_DEBUG_TRANSACTION_COMPLETE,
                                 "binder: %d:%d BR_TRANSACTION_COMPLETE\n",
                                 proc->pid, thread->pid);
//将 w 从 thread->todo 删除
                    list_del(&w->entry);
                    kfree(w);
                    binder_stats_deleted(BINDER_STAT_TRANSACTION_COMPLETE);
            } break;
            case BINDER_WORK_NODE: {
                    struct binder_node *node = container_of(w, struct binder_node, work);
                    uint32_t cmd = BR_NOOP;
                    const char *cmd_name;
                    int strong = node->internal_strong_refs || node->local_strong_refs;
                    int weak = !hlist_empty(&node->refs) || node->local_weak_refs || strong;
                    if (weak && !node->has_weak_ref) {
                            cmd = BR_INCREFS;
                            cmd_name = "BR_INCREFS";
                            node->has_weak_ref = 1;
                            node->pending_weak_ref = 1;
                            node->local_weak_refs++;
                    } else if (strong && !node->has_strong_ref) {
                            cmd = BR_ACQUIRE;
                            cmd_name = "BR_ACQUIRE";
                            node->has_strong_ref = 1;
                            node->pending_strong_ref = 1;
                            node->local_strong_refs++;
                    } else if (!strong && node->has_strong_ref) {
                            cmd = BR_RELEASE;
                            cmd_name = "BR_RELEASE";
                            node->has_strong_ref = 0;
                    } else if (!weak && node->has_weak_ref) {
                            cmd = BR_DECREFS;
                            cmd_name = "BR_DECREFS";
                            node->has_weak_ref = 0;
                    }
                    if (cmd != BR_NOOP) {
```

```
                        if (put_user(cmd, (uint32_t __user *)ptr))
                            return -EFAULT;
                        ptr += sizeof(uint32_t);
                        if (put_user(node->ptr, (void * __user *)ptr))
                            return -EFAULT;
                        ptr += sizeof(void *);
                        if (put_user(node->cookie, (void * __user *)ptr))
                            return -EFAULT;
                        ptr += sizeof(void *);
                        binder_stat_br(proc, thread, cmd);
                        binder_debug(BINDER_DEBUG_USER_REFS,
                                    "binder: %d:%d %s %d u%p c%p\n",
                                    proc->pid, thread->pid, cmd_name, node->debug_id, node->
ptr, node->cookie);
                    } else {
                        list_del_init(&w->entry);
                        if (!weak && !strong) {
                            binder_debug(BINDER_DEBUG_INTERNAL_REFS,
                                        "binder: %d:%d node %d u%p c%p deleted\n",
                                        proc->pid, thread->pid, node->debug_id,
                                        node->ptr, node->cookie);
                            rb_erase(&node->rb_node, &proc->nodes);
                            kfree(node);
                            binder_stats_deleted(BINDER_STAT_NODE);
                        } else {
                            binder_debug(BINDER_DEBUG_INTERNAL_REFS,
                                        "binder: %d:%d node %d u%p c%p state unchanged\n",
                                        proc->pid, thread->pid, node->debug_id, node->ptr,
                                        node->cookie);
                        }
                    }
                } break;
                case BINDER_WORK_DEAD_BINDER:
                case BINDER_WORK_DEAD_BINDER_AND_CLEAR:
                case BINDER_WORK_CLEAR_DEATH_NOTIFICATION: {
                    struct binder_ref_death *death;
                    uint32_t cmd;
                    death = container_of(w, struct binder_ref_death, work);
                    if (w->type == BINDER_WORK_CLEAR_DEATH_NOTIFICATION)
                        cmd = BR_CLEAR_DEATH_NOTIFICATION_DONE;
                    else
                        cmd = BR_DEAD_BINDER;
                    if (put_user(cmd, (uint32_t __user *)ptr))
                        return -EFAULT;
                    ptr += sizeof(uint32_t);
                    if (put_user(death->cookie, (void * __user *)ptr))
                        return -EFAULT;
                    ptr += sizeof(void *);
                    binder_stat_br(proc, thread, cmd);
                    binder_debug(BINDER_DEBUG_DEATH_NOTIFICATION,
                                "binder: %d:%d %s %p\n",
```

```
                        proc->pid, thread->pid,
                        cmd == BR_DEAD_BINDER ?
                        "BR_DEAD_BINDER" :
                        "BR_CLEAR_DEATH_NOTIFICATION_DONE",
                        death->cookie);
            if (w->type == BINDER_WORK_CLEAR_DEATH_NOTIFICATION) {
                list_del(&w->entry);
                kfree(death);
                binder_stats_deleted(BINDER_STAT_DEATH);
            } else
                list_move(&w->entry, &proc->delivered_death);
            if (cmd == BR_DEAD_BINDER)
                goto done; /* DEAD_BINDER notifications can cause transactions */
        } break;
        }
        if (!t)
            continue;
        BUG_ON(t->buffer == NULL);
//把事务项 t 中的数据复制到本地局部变量 struct binder_transaction_data tr 中
        if (t->buffer->target_node) {
            struct binder_node *target_node = t->buffer->target_node;
            tr.target.ptr = target_node->ptr;
            tr.cookie =   target_node->cookie;
            t->saved_priority = task_nice(current);
            if (t->priority < target_node->min_priority &&
                !(t->flags & TF_ONE_WAY))
                binder_set_nice(t->priority);
            else if (!(t->flags & TF_ONE_WAY) ||
                 t->saved_priority > target_node->min_priority)
                binder_set_nice(target_node->min_priority);
            cmd = BR_TRANSACTION;
        } else {
            tr.target.ptr = NULL;
            tr.cookie = NULL;
            cmd = BR_REPLY;
        }
        tr.code = t->code;
        tr.flags = t->flags;
        tr.sender_euid = t->sender_euid;
        if (t->from) {
            struct task_struct *sender = t->from->proc->tsk;
            tr.sender_pid = task_tgid_nr_ns(sender,
                                current->nsproxy->pid_ns);
        } else {
            tr.sender_pid = 0;
        }
        tr.data_size = t->buffer->data_size;
        tr.offsets_size = t->buffer->offsets_size;
//t->buffer->data 所指向的地址是内核空间的，如果要把数据返回给 Service Manager 进程的用户空间
//而 Service Manager 进程的用户空间是不能访问内核空间的数据的，所以需要进一步处理
//在具体处理时，Binder 机制使用类似浅拷贝的方法，通过在用户空间分配一个虚拟地址
```

```
//然后让这个用户空间虚拟地址与 t->buffer->data 这个内核空间虚拟地址指向同一个物理地址
//在此只需将 t->buffer->data 加上一个偏移值 proc->user_buffer_offset
//就可以得到 t->buffer->data 对应的用户空间虚拟地址了
//在调整了 tr.data.ptr.buffer 值后，需要一起调整 tr.data.ptr.offsets 的值
                tr.data.ptr.buffer = (void *)t->buffer->data +
                                        proc->user_buffer_offset;
                tr.data.ptr.offsets = tr.data.ptr.buffer +
                                        ALIGN(t->buffer->data_size,
                                          sizeof(void *));
//把 tr 的内容复制到用户传进来的缓冲区，指针 ptr 指向这个用户缓冲区的地址
                if (put_user(cmd, (uint32_t __user *)ptr))
                        return -EFAULT;
                ptr += sizeof(uint32_t);
                if (copy_to_user(ptr, &tr, sizeof(tr)))
                        return -EFAULT;
                ptr += sizeof(tr);
//上述代码只是对 tr.data.ptr.bufferr 和 tr.data.ptr.offsets 的内容进行了浅拷贝工作
                trace_binder_transaction_received(t);
                binder_stat_br(proc, thread, cmd);
                binder_debug(BINDER_DEBUG_TRANSACTION,
                             "binder: %d:%d %s %d %d:%d, cmd %d"
                             "size %zd-%zd ptr %p-%p\n",
                             proc->pid, thread->pid,
                             (cmd == BR_TRANSACTION) ? "BR_TRANSACTION" :
                             "BR_REPLY",
                             t->debug_id, t->from ? t->from->proc->pid : 0,
                             t->from ? t->from->pid : 0, cmd,
                             t->buffer->data_size, t->buffer->offsets_size,
                             tr.data.ptr.buffer, tr.data.ptr.offsets);
//从 todo 列表中删除，因为已经处理了这个事务
                list_del(&t->work.entry);
                t->buffer->allow_user_free = 1;
//如果 cmd == BR_TRANSACTION && !(t->flags & TF_ONE_WAY)为 true
//说明虽然在驱动程序中已经处理完了这个事务，但是仍然要在 Service Manager 完成之后等待回复确认
                if (cmd == BR_TRANSACTION && !(t->flags & TF_ONE_WAY)) {
//把当前事务 t 放在 thread->transaction_stack 队列的头部
                        t->to_parent = thread->transaction_stack;
                        t->to_thread = thread;
                        thread->transaction_stack = t;
                }
//如果 cmd == BR_TRANSACTION && !(t->flags & TF_ONE_WAY)为 false
//则不需要等待回复了，而是直接删除事务 t
                    else {
                        t->buffer->transaction = NULL;
                        kfree(t);
                        binder_stats_deleted(BINDER_STAT_TRANSACTION);
                }
                break;
        }
done:
        *consumed = ptr - buffer;
```

```
        if (proc->requested_threads + proc->ready_threads == 0 &&
            proc->requested_threads_started < proc->max_threads &&
            (thread->looper & (BINDER_LOOPER_STATE_REGISTERED |
             BINDER_LOOPER_STATE_ENTERED)) /* the user-space code fails to */
             /*spawn a new thread if we leave this out */) {
                proc->requested_threads++;
                binder_debug(BINDER_DEBUG_THREADS,
                             "binder: %d:%d BR_SPAWN_LOOPER\n",
                             proc->pid, thread->pid);
                if (put_user(BR_SPAWN_LOOPER, (uint32_t __user *)buffer))
                        return -EFAULT;
                binder_stat_br(proc, thread, BR_SPAWN_LOOPER);
        }
        return 0;
}
```

由此可见，Binder 驱动程序是通过如下 4 个协议来引用运行在 Server 进程中的 Binder 本地对象的。

- ☑ BR_INCREFS
- ☑ BR_ACQUIRE
- ☑ BR_DECREFS
- ☑ BR_RELEASE

5.5.2 处理接口协议

在文件 frameworks/native/libs/binder/IPCThreadState.cpp 中，通过使用类成员函数 executeCommand 来处理 5.5.1 节中介绍的 4 个接口协议。函数 executeCommand()的具体实现代码如下所示。

```
status_t IPCThreadState::executeCommand(int32_t cmd)
{
    BBinder* obj;
    RefBase::weakref_type* refs;
    status_t result = NO_ERROR;

    switch (cmd) {
    case BR_ERROR:
        result = mIn.readInt32();
        break;

    case BR_OK:
        break;

    case BR_ACQUIRE:
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();
        ALOG_ASSERT(refs->refBase() == obj,
                    "BR_ACQUIRE: object %p does not match cookie %p (expected %p)",
                    refs, obj, refs->refBase());
        obj->incStrong(mProcess.get());
        IF_LOG_REMOTEREFS() {
```

```
            LOG_REMOTEREFS("BR_ACQUIRE from driver on %p", obj);
            obj->printRefs();
        }
        mOut.writeInt32(BC_ACQUIRE_DONE);
        mOut.writeInt32((int32_t)refs);
        mOut.writeInt32((int32_t)obj);
        break;

    case BR_RELEASE:
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();
        ALOG_ASSERT(refs->refBase() == obj,
                    "BR_RELEASE: object %p does not match cookie %p (expected %p)",
                    refs, obj, refs->refBase());
        IF_LOG_REMOTEREFS() {
            LOG_REMOTEREFS("BR_RELEASE from driver on %p", obj);
            obj->printRefs();
        }
        mPendingStrongDerefs.push(obj);
        break;

    case BR_INCREFS:
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();
        refs->incWeak(mProcess.get());
        mOut.writeInt32(BC_INCREFS_DONE);
        mOut.writeInt32((int32_t)refs);
        mOut.writeInt32((int32_t)obj);
        break;

    case BR_DECREFS:
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();
        mPendingWeakDerefs.push(refs);
        break;

    case BR_ATTEMPT_ACQUIRE:
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();

        {
            const bool success = refs->attemptIncStrong(mProcess.get());
            ALOG_ASSERT(success && refs->refBase() == obj,
                        "BR_ATTEMPT_ACQUIRE: object %p does not match cookie %p (expected %p)",
                        refs, obj, refs->refBase());

            mOut.writeInt32(BC_ACQUIRE_RESULT);
            mOut.writeInt32((int32_t)success);
        }
        break;
```

```
case BR_TRANSACTION:
    {
        binder_transaction_data tr;
        result = mIn.read(&tr, sizeof(tr));
        ALOG_ASSERT(result == NO_ERROR,
            "Not enough command data for brTRANSACTION");
        if (result != NO_ERROR) break;

        Parcel buffer;
        buffer.ipcSetDataReference(
            reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer),
            tr.data_size,
            reinterpret_cast<const size_t*>(tr.data.ptr.offsets),
            tr.offsets_size/sizeof(size_t), freeBuffer, this);

        const pid_t origPid = mCallingPid;
        const uid_t origUid = mCallingUid;

        mCallingPid = tr.sender_pid;
        mCallingUid = tr.sender_euid;

        int curPrio = getpriority(PRIO_PROCESS, mMyThreadId);
        if (gDisableBackgroundScheduling) {
            if (curPrio > ANDROID_PRIORITY_NORMAL) {
                setpriority(PRIO_PROCESS, mMyThreadId, ANDROID_PRIORITY_NORMAL);
            }
        } else {
            if (curPrio >= ANDROID_PRIORITY_BACKGROUND) {
                set_sched_policy(mMyThreadId, SP_BACKGROUND);
            }
        }

        //ALOGI(">>>> TRANSACT from pid %d uid %d\n", mCallingPid, mCallingUid);

        Parcel reply;
        IF_LOG_TRANSACTIONS() {
            TextOutput::Bundle _b(alog);
            alog << "BR_TRANSACTION thr " << (void*)pthread_self()
                << " / obj " << tr.target.ptr << " / code "
                << TypeCode(tr.code) << ": " << indent << buffer
                << dedent << endl
                << "Data addr = "
                << reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer)
                << ", offsets addr="
                << reinterpret_cast<const size_t*>(tr.data.ptr.offsets) << endl;
        }
        if (tr.target.ptr) {
            sp<BBinder> b((BBinder*)tr.cookie);
            const status_t error = b->transact(tr.code, buffer, &reply, tr.flags);
            if (error < NO_ERROR) reply.setError(error);
```

```
            } else {
                const status_t error = the_context_object->transact(tr.code, buffer, &reply, tr.flags);
                if (error < NO_ERROR) reply.setError(error);
            }

            //ALOGI("<<<< TRANSACT from pid %d restore pid %d uid %d\n",
            //      mCallingPid, origPid, origUid);

            if ((tr.flags & TF_ONE_WAY) == 0) {
                LOG_ONEWAY("Sending reply to %d!", mCallingPid);
                sendReply(reply, 0);
            } else {
                LOG_ONEWAY("NOT sending reply to %d!", mCallingPid);
            }

            mCallingPid = origPid;
            mCallingUid = origUid;

            IF_LOG_TRANSACTIONS() {
                TextOutput::Bundle _b(alog);
                alog << "BC_REPLY thr " << (void*)pthread_self() << " / obj "
                    << tr.target.ptr << ": " << indent << reply << dedent << endl;
            }

        }
        break;

    case BR_DEAD_BINDER:
        {
            BpBinder *proxy = (BpBinder*)mIn.readInt32();
            proxy->sendObituary();
            mOut.writeInt32(BC_DEAD_BINDER_DONE);
            mOut.writeInt32((int32_t)proxy);
        } break;

    case BR_CLEAR_DEATH_NOTIFICATION_DONE:
        {
            BpBinder *proxy = (BpBinder*)mIn.readInt32();
            proxy->getWeakRefs()->decWeak(proxy);
        } break;

    case BR_FINISHED:
        result = TIMED_OUT;
        break;

    case BR_NOOP:
        break;

    case BR_SPAWN_LOOPER:
        mProcess->spawnPooledThread(false);
        break;
```

```
    default:
        printf("*** BAD COMMAND %d received from Binder driver\n", cmd);
        result = UNKNOWN_ERROR;
        break;
    }
    if (result != NO_ERROR) {
        mLastError = result;
    }
    return result;
}
```

通过上述代码可知，在函数 executeCommand()中会调用 BBinder::transact 来处理 Client 端的请求。当需要多个线程提供服务时，驱动会请求创建新线程。具体创建某线程的过程，可以通过上述函数处理 BR_SPAWN_LOOPER 协议的过程获得。

5.6　引用对象 binder_ref

在 Android 系统中，引用对象的类型为 binder_ref，定义结构体 binder_ref 的代码如下所示。

```
struct binder_ref {
    //调试 id
    int debug_id;
    //挂载到宿主对象 binder_proc 的红黑树 refs_by_desc 中的节点
    struct rb_node rb_node_desc;
    //挂载到宿主对象 binder_proc 的红黑树 refs_by_node 中的节点
    struct rb_node rb_node_node;
    //挂载到 Binder 实体对象的 refs 链表中的节点
    struct hlist_node node_entry;
    //Binder 引用对象的宿主进程 binder_proc
    struct binder_proc *proc;
    //Binder 引用对象所引用的 Binder 实体对象
    struct binder_node *node;
    //Binder 引用对象的句柄值
    uint32_t desc;
    //强引用计数
    int strong;
    //弱引用计数
    int weak;
    //注册死亡接收通知
    struct binder_ref_death *death;
};
```

在 Binder 机制中，binder_ref 用来描述一个 Binder 引用对象，每一个 client 在 Binder 驱动中都有一个 binder 引用对象。各个成员变量的具体说明如下所示。

☑ 成员变量 node：保存了该 Binder 引用对象所引用的 Binder 实体对象，Binder 实体对象使用链表保存了所有引用该实体对象的 Binder 引用对象。

☑ node_entry：是该 Binder 引用对象所引用的实体对象的成员变量 refs 链表中的节点。
☑ desc：是一个句柄值，用来描述一个 Binder 引用对象。
☑ node：当 Client 进程通过句柄值来访问某个 Service 时，Binder 驱动程序可以通过该句柄值找到对应的 Binder 引用对象，然后根据该 Binder 引用对象的成员变量 node 找到对应的 Binder 实体对象，最后通过该 Binder 实体对象找到要访问的 Service。
☑ proc：执行该 Binder 引用对象的宿主进程。
☑ rb_node_desc 和 rb_node_node：是 binder_proc 中红黑树 refs_by_desc 和 refs_by_node 的节点。

Binder 驱动程序存在如下 4 个重要的协议，用于增加和减少 Binder 引用对象的强引用计数和弱引用计数。

☑ BR_INCREFS
☑ BR_ACQUIRE
☑ BR_DECREFS
☑ BR_RELEASE

上述计数处理功能通过函数 binder_thread_write()实现，此函数在文件/drivers/staging/android/binder.c 中定义，此函数已经在本章前面的内容中进行了详细分析。

协议 BR_INCREFS 和 BR_ACQUIRE 用于增加一个 Binder 引用对象的强引用计数和弱引用计数，此功能是通过调用函数 binder_inc_ref()实现的，具体实现代码如下所示。

```
static int binder_inc_ref(struct binder_ref *ref, int strong,
                          struct list_head *target_list)
{
        int ret;
        if (strong) {
                if (ref->strong == 0) {
                        ret = binder_inc_node(ref->node, 1, 1, target_list);
                        if (ret)
                                return ret;
                }
                ref->strong++;
        } else {
                if (ref->weak == 0) {
                        ret = binder_inc_node(ref->node, 0, 1, target_list);
                        if (ret)
                                return ret;
                }
                ref->weak++;
        }
        return 0;
}
```

而协议 BR_RELEASE 和 BR_DECREFS 用于减少一个 Binder 引用对象的强引用计数和弱引用计数，此功能是通过调用函数 binder_dec_ref()实现的，具体实现代码如下所示。

```
static int binder_dec_ref(struct binder_ref *ref, int strong)
{
        if (strong) {
```



```
				if (ref->strong == 0) {
					binder_user_error("binder: %d invalid dec strong, "
							"ref %d desc %d s %d w %d\n",
							ref->proc->pid, ref->debug_id,
							ref->desc, ref->strong, ref->weak);
					return -EINVAL;
				}
				ref->strong--;
				if (ref->strong == 0) {
					int ret;
					ret = binder_dec_node(ref->node, strong, 1);
					if (ret)
						return ret;
				}
		} else {
				if (ref->weak == 0) {
					binder_user_error("binder: %d invalid dec weak, "
							"ref %d desc %d s %d w %d\n",
							ref->proc->pid, ref->debug_id,
							ref->desc, ref->strong, ref->weak);
					return -EINVAL;
				}
				ref->weak--;
		}
		if (ref->strong == 0 && ref->weak == 0)
				binder_delete_ref(ref);
		return 0;
}
```

再看函数 binder_delete_ref()，功能是销毁 binder_ref 对象，具体实现代码如下所示。

```
static void binder_delete_ref(struct binder_ref *ref)
{
		binder_debug(BINDER_DEBUG_INTERNAL_REFS,
				"binder: %d delete ref %d desc %d for "
				"node %d\n", ref->proc->pid, ref->debug_id,
				ref->desc, ref->node->debug_id);

		rb_erase(&ref->rb_node_desc, &ref->proc->refs_by_desc);
		rb_erase(&ref->rb_node_node, &ref->proc->refs_by_node);
		if (ref->strong)
				binder_dec_node(ref->node, 1, 1);
		hlist_del(&ref->node_entry);
		binder_dec_node(ref->node, 0, 1);
		if (ref->death) {
				binder_debug(BINDER_DEBUG_DEAD_BINDER,
						"binder: %d delete ref %d desc %d "
						"has death notification\n", ref->proc->pid,
						ref->debug_id, ref->desc);
				list_del(&ref->death->work.entry);
				kfree(ref->death);
```

```
                binder_stats_deleted(BINDER_STAT_DEATH);
        }
        kfree(ref);
        binder_stats_deleted(BINDER_STAT_REF);
}
```

5.7 代理对象 BpBinder

在 Android 系统中，代理对象 BpBinder 是远程对象在当前进程的代理，它实现了 IBinder 接口。对于初学者来说，BBinder 与 BpBinder 这两者容易混淆。其实这两者很好区分，对于 Service 来说继承了 BBinder(BnInterface)，因为 BBinder 有 onTransact()消息处理函数。而对于与 Service 通信的 Client 来说，需要继承 BpBinder(BpInterface)，因为 BpBinder 有消息传递函数 transact()。本节将详细讲解 Android 5.0 代理对象 BpBinder 的核心知识。

5.7.1 创建 Binder 代理对象

以 cameraService 的 client 为例，文件 Camera.cpp 中的函数 getCameraService()能够获取远程 CameraService 的 IBinder 对象，然后通过如下代码进行了重构，得到了 BpCameraService 对象。

```
mCameraService = interface_cast<ICameraService>(binder);
```

而 BpCameraService 继承了 BpInterface，并传入了 BBinder。

```
cameraService：
    defaultServiceManager()->addService(
            String16("media.camera"), new CameraService());
```

在 IPC 传递的过程中，IBinder 指针不可缺少。这个指针对一个进程来说就像是 socket 的 ID 一样，是唯一的。无论这个 IBinder 是 BBinder 还是 BpBinder，它们都是在重构 BpBinder 或者 BBinder 时把 IBinder 作为参数传入的。

在 Android 系统中，创建 Binder 代理对象的方法在文件 frameworks/native/libs/binder/BpBinder.cpp 中定义，具体实现代码如下所示。

```
BpBinder::BpBinder(int32_t handle)
    : mHandle(handle)
    , mAlive(1)
    , mObitsSent(0)
    , mObituaries(NULL)
{
    ALOGV("Creating BpBinder %p handle %d\n", this, mHandle);
    //设置 Binder 代理对象的生命周期收到弱引用的计数影响
    extendObjectLifetime(OBJECT_LIFETIME_WEAK);
    //调用当前线程内部的 IPCThreadState 的成员函数 incWeakHandle()增加相应的 Binder 引用对象的弱引用计数
    IPCThreadState::self()->incWeakHandle(handle);
}
```

在文件 frameworks/native/libs/binder/IPCThreadState.cpp 中，定义成员函数 incWeakHandle()的代码如下所示。

```
void IPCThreadState::incWeakHandle(int32_t handle)
{
    LOG_REMOTEREFS("IPCThreadState::incWeakHandle(%d)\n", handle);
    mOut.writeInt32(BC_INCREFS);
    mOut.writeInt32(handle);
}
```

5.7.2　销毁 Binder 代理对象

当销毁一个 Binder 代理对象时，线程会调用内部的 IPCThreadState 对象的成员函数 decWeakHandle()来减少相应的 Binder 引用对象的弱引用计数。函数 decWeakHandle()的具体实现代码如下所示。

```
void IPCThreadState::decWeakHandle(int32_t handle)
{
    LOG_REMOTEREFS("IPCThreadState::decWeakHandle(%d)\n", handle);
    mOut.writeInt32(BC_DECREFS);
    mOut.writeInt32(handle);
}
```

在 Binder 代理对象中，其 transact()函数的实现代码如下所示。

```
status_t BpBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    if (mAlive) {
        status_t status = IPCThreadState::self()->transact(
            mHandle, code, data, reply, flags);
        if (status == DEAD_OBJECT) mAlive = 0;
        return status;
    }
    return DEAD_OBJECT;
}
```

各个参数的具体说明如下所示。

☑　code：表示请求的 ID 号。

☑　data：表示请求的参数。

☑　reply：表示返回的结果。

☑　flags：表示一些额外的标识，如 FLAG_ONEWAY，通常为 0。

上述 transact()函数只是简单调用了 IPCThreadState::self()的 transact， IPCThreadState::transact 中的定义代码如下所示。

```
status_t IPCThreadState::transact(int32_t handle,
                                  uint32_t code, const Parcel& data,
                                  Parcel* reply, uint32_t flags)
{
```

```
    status_t err = data.errorCheck();

    flags |= TF_ACCEPT_FDS;

    IF_LOG_TRANSACTIONS() {
        TextOutput::Bundle _b(alog);
        alog << "BC_TRANSACTION thr " << (void*)pthread_self() << " / hand "
            << handle << " / code " << TypeCode(code) << ": "
            << indent << data << dedent << endl;
    }

    if (err == NO_ERROR) {
        LOG_ONEWAY(">>>> SEND from pid %d uid %d %s", getpid(), getuid(),
            (flags & TF_ONE_WAY) == 0 ? "READ REPLY" : "ONE WAY");
        err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);
    }

    if (err != NO_ERROR) {
        if (reply) reply->setError(err);
        return (mLastError = err);
    }

    if ((flags & TF_ONE_WAY) == 0) {
        if (reply) {
            err = waitForResponse(reply);
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }

        IF_LOG_TRANSACTIONS() {
            TextOutput::Bundle _b(alog);
            alog << "BR_REPLY thr " << (void*)pthread_self() << " / hand "
                << handle << ": ";
            if (reply) alog << indent << *reply << dedent << endl;
            else alog << "(none requested)" << endl;
        }
    } else {
        err = waitForResponse(NULL, NULL);
    }

    return err;
}

status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
{
    int32_t cmd;
    int32_t err;

    while (1) {
        if ((err=talkWithDriver()) < NO_ERROR) break;
```

```
err = mIn.errorCheck();
if (err < NO_ERROR) break;
if (mIn.dataAvail() == 0) continue;

cmd = mIn.readInt32();

IF_LOG_COMMANDS() {
    alog << "Processing waitForResponse Command: "
        << getReturnString(cmd) << endl;
}

switch (cmd) {
case BR_TRANSACTION_COMPLETE:
    if (!reply && !acquireResult) goto finish;
    break;

case BR_DEAD_REPLY:
    err = DEAD_OBJECT;
    goto finish;

case BR_FAILED_REPLY:
    err = FAILED_TRANSACTION;
    goto finish;

case BR_ACQUIRE_RESULT:
    {
        LOG_ASSERT(acquireResult != NULL, "Unexpected brACQUIRE_RESULT");
        const int32_t result = mIn.readInt32();
        if (!acquireResult) continue;
        *acquireResult = result ? NO_ERROR : INVALID_OPERATION;
    }
    goto finish;

case BR_REPLY:
    {
        binder_transaction_data tr;
        err = mIn.read(&tr, sizeof(tr));
        LOG_ASSERT(err == NO_ERROR, "Not enough command data for brREPLY");
        if (err != NO_ERROR) goto finish;

        if (reply) {
            if ((tr.flags & TF_STATUS_CODE) == 0) {
                reply->ipcSetDataReference(
                    reinterpret_cast(tr.data.ptr.buffer),
                    tr.data_size,
                    reinterpret_cast(tr.data.ptr.offsets),
                    tr.offsets_size/sizeof(size_t),
                    freeBuffer, this);
            } else {
                err = *static_cast(tr.data.ptr.buffer);
                freeBuffer(NULL,
```

```
                                        reinterpret_cast(tr.data.ptr.buffer),
                                        tr.data_size,
                                        reinterpret_cast(tr.data.ptr.offsets),
                                        tr.offsets_size/sizeof(size_t), this);
                            }
                        } else {
                            freeBuffer(NULL,
                                reinterpret_cast(tr.data.ptr.buffer),
                                tr.data_size,
                                reinterpret_cast(tr.data.ptr.offsets),
                                tr.offsets_size/sizeof(size_t), this);
                            continue;
                        }
                    }
                    goto finish;

            default:
                    err = executeCommand(cmd);
                    if (err != NO_ERROR) goto finish;
                    break;
            }
        }

finish:
    if (err != NO_ERROR) {
        if (acquireResult) *acquireResult = err;
        if (reply) reply->setError(err);
        mLastError = err;
    }

    return err;
}
```

在上述代码中，通过内核模块 transact 将请求发送给服务端。当服务端处理完请求之后，会沿着原路返回结果给调用者。在此可以看出请求是同步操作，它会等待直到结果返回为止。这样在 BpBinder 之上进行简单包装之后，就可以得到与服务对象相同的接口，调用者无须关心调用的对象是远程的还是本地的。

第 6 章　init 启动进程详解

在运行 Android 程序后首先会启动 init 进程，此进程是 Linux 系统中用户空间的第一个进程，进程编号为 1。本章将详细讲解在 Android 5.0 系统中启动 init 进程的知识，希望通过本章内容的学习，为后面高级应用的知识打下基础。

6.1　什么是 init 进程

Android 是一个基于 Linux 内核的系统，与 Linux、Fedora Linux 最大的区别是，Android 在应用层专门为移动设备添加了一些特有的支持。目前 Linux 有很多通信机制可以在用户空间和内核空间之间交互，例如设备驱动文件（位于/dev 目录中）、内存文件（/proc、/sys 目录等）。Android 在加载 Linux 基本内核后，就开始运行一个初始化进程 init。从 Android 加载 Linux 内核时设置了如下参数。

```
Kernelcommand line: noinitrd root=/dev/nfs console=ttySAC0 init=/initnfsroot=192.168.1.103:/nfsbootip=
192.168.1.20:192.168.1.103:192.168.1.1:255.255.255.0::eth0:on
```

在上述命令中，告诉 Linux 内核初始化完成后开始运行 init 进程，由于 init 进程就是放在系统根目录下，而 init 进程的代码位于源码的目录 system/core/init，在分析 init 的核心代码之前，还需要做如下工作。

- ☑ 初始化属性。
- ☑ 处理配置文件的命令（主要是 init.rc 文件），包括处理各种 Action。
- ☑ 性能分析（使用 bootchart 工具）。
- ☑ 无限循环执行 command（启动其他的进程）。

init 程序并不是由一个源代码文件组成的，而是由一组源代码文件的目标文件链接而成的。这些文件位于目录/system/core/init 中。

主要的 JNI 代码放在路径 frameworks/base/core/jni/中。

另外，还涉及了其他目录中的如下文件。

- ☑ /bionic/libc/bionic/libc_init_common.h
- ☑ /bionic/libc/bionic/libc_init_common.c
- ☑ /bionic/libc/bionic/libc_init_dynamic.c
- ☑ /bionic/libc/bionic/libc_init_static.c
- ☑ /system/core/libcutils/properties.c

本章将仔细地分析 init 进程的启动过程，了解 Android 系统是如何启动的。

6.2 入口函数

在 Android 5.0 系统中，进程 init 入口函数是 main()，其实现文件的路径为/system/core/init/init.c。函数 main 的具体实现代码如下所示。

```
int main(int argc, char **argv)
{
        int fd_count = 0;
        struct pollfd ufds[4];
        char *tmpdev;
        char* debuggable;
        char tmp[32];
        int property_set_fd_init = 0;
        int signal_fd_init = 0;
        int keychord_fd_init = 0;
        bool is_charger = false;

        if (!strcmp(basename(argv[0]), "ueventd"))
                return ueventd_main(argc, argv);

        if (!strcmp(basename(argv[0]), "watchdogd"))
                return watchdogd_main(argc, argv);

        /* clear the umask */
        umask(0);
        //下面的代码开始建立各种用户空间的目录，如/dev、/proc、/sys 等
        mkdir("/dev", 0755);
        mkdir("/proc", 0755);
        mkdir("/sys", 0755);

        mount("tmpfs", "/dev", "tmpfs", MS_NOSUID, "mode=0755");
        mkdir("/dev/pts", 0755);
        mkdir("/dev/socket", 0755);
        mount("devpts", "/dev/pts", "devpts", 0, NULL);
        mount("proc", "/proc", "proc", 0, NULL);
        mount("sysfs", "/sys", "sysfs", 0, NULL);

        /*  检测/dev/.booting 文件是否可读写和创建*/
        close(open("/dev/.booting", O_WRONLY | O_CREAT, 0000));

        open_devnull_stdio();
        klog_init();
        //初始化属性
        property_init();

        get_hardware_name(hardware, &revision);
        //处理内核命令行
```

```
    process_kernel_cmdline();
    ……

    is_charger = !strcmp(bootmode, "charger");

    INFO("property init\n");
    if (!is_charger)
        property_load_boot_defaults();

    INFO("reading config file\n");
    //分析/init.rc 文件的内容
    init_parse_config_file("/init.rc");
    …//执行初始化文件中的动作
    action_for_each_trigger("init", action_add_queue_tail);
    //在 charger 模式下略过 mount 文件系统的工作
    if (!is_charger) {
        action_for_each_trigger("early-fs", action_add_queue_tail);
        action_for_each_trigger("fs", action_add_queue_tail);
        action_for_each_trigger("post-fs", action_add_queue_tail);
        action_for_each_trigger("post-fs-data", action_add_queue_tail);
    }

    queue_builtin_action(property_service_init_action, "property_service_init");
    queue_builtin_action(signal_init_action, "signal_init");
    queue_builtin_action(check_startup_action, "check_startup");

    if (is_charger) {
        action_for_each_trigger("charger", action_add_queue_tail);
    } else {
        action_for_each_trigger("early-boot", action_add_queue_tail);
        action_for_each_trigger("boot", action_add_queue_tail);
    }

        /* run all property triggers based on current state of the properties */
    queue_builtin_action(queue_property_triggers_action, "queue_property_triggers");


#if BOOTCHART
    queue_builtin_action(bootchart_init_action, "bootchart_init");
#endif
    //进入无限循环，建立 init 的子进程（init 是所有进程的父进程）
    for(;;) {
        int nr, i, timeout = -1;
        //执行命令（子进程对应的命令）
        execute_one_command();
        restart_processes();

        if (!property_set_fd_init && get_property_set_fd() > 0) {
            ufds[fd_count].fd = get_property_set_fd();
            ufds[fd_count].events = POLLIN;
            ufds[fd_count].revents = 0;
```

```
            fd_count++;
            property_set_fd_init = 1;
        }
        if (!signal_fd_init && get_signal_fd() > 0) {
            ufds[fd_count].fd = get_signal_fd();
            ufds[fd_count].events = POLLIN;
            ufds[fd_count].revents = 0;
            fd_count++;
            signal_fd_init = 1;
        }
        if (!keychord_fd_init && get_keychord_fd() > 0) {
            ufds[fd_count].fd = get_keychord_fd();
            ufds[fd_count].events = POLLIN;
            ufds[fd_count].revents = 0;
            fd_count++;
            keychord_fd_init = 1;
        }

        if (process_needs_restart) {
            timeout = (process_needs_restart - gettime()) * 1000;
            if (timeout < 0)
                timeout = 0;
        }

        if (!action_queue_empty() || cur_action)
            timeout = 0;
//bootchart 是一个性能统计工具，用于搜集硬件和系统的信息，并将其写入磁盘，以便其他程序使用
#if BOOTCHART
        if (bootchart_count > 0) {
            if (timeout < 0 || timeout > BOOTCHART_POLLING_MS)
                timeout = BOOTCHART_POLLING_MS;
            if (bootchart_step() < 0 || --bootchart_count == 0) {
                bootchart_finish();
                bootchart_count = 0;
            }
        }
#endif
        //等待下一个命令的提交
        nr = poll(ufds, fd_count, timeout);
        if (nr <= 0)
            continue;

        for (i = 0; i < fd_count; i++) {
            if (ufds[i].revents == POLLIN) {
                if (ufds[i].fd == get_property_set_fd())
                    handle_property_set_fd();
                else if (ufds[i].fd == get_keychord_fd())
                    handle_keychord();
                else if (ufds[i].fd == get_signal_fd())
                    handle_signal();
            }
```

```
        }
    }

    return 0;
}
```

从上述 main()函数的实现代码中可以看出，init 进程分为如下两个部分。

（1）初始化工作，主要包括建立/dev、/proc 等目录，初始化属性，执行 init.rc 等初始化文件中的 action 等。

（2）使用 for 循环建立子进程。

在 Linux 系统中，init 是一切应用空间进程的父进程。所以平常在 Linux 终端执行的命令，并建立进程，实际上都是在这个无限的 for 循环中完成的。也就是说，在 Linux 终端执行 ps －e 命令后，看到的所有除了 init 外的其他进程，都是由 init 负责创建的。而且 init 也会常驻内容。当然，如果 init 崩溃了，那么 Linux 系统基本上也就崩溃了。

6.3　init 配置文件

在 Android 5.0 系统的 init 进程中，配置文件 init.rc 的路径是/system/core/rootdir/init.rc，本节将详细讲解配置文件 init.rc 的基本知识。

6.3.1　init.rc 基础

在 Android 5.0 系统中，文件 init.rc 是一个可配置的初始化文件，其中定制了厂商可以配置的额外的初始化配置信息，具体格式如下：

```
init.%PRODUCT%.rc
```

文件 init.rc 是在文件/system/core/init/init.c 中读取的，它基于“行”，包含一些用空格隔开的关键字（它属于特殊字符）。如果在关键字中含有空格，则使用“/”表示转义，使用“　”防止关键字被断开，如果“/”在末尾则表示换行，以“#”开头的表示注释。

文件 init.rc 包含 4 种状态类别，分别是 Actions、Commands、Services 和 Options，当声明一个 Service 或者 Action 时，将隐式声明一个 section，它之后跟随的 Command 或者 Option 都将属于这个 section。另外，Action 和 Service 不能重名，否则忽略为 error。

（1）Actions

Actions 就是在某种条件下触发一系列的命令，通常有一个 trigger，例如：

```
on <trigger>
        <command>
        <command>
```

（2）Service

Service 的结构如下所示。

```
service <name> <pathname> [ <argument> ]*
    <option>
    <option>
```

（3）Option

Option 是 Service 的修饰词，主要包括如下选项。

- ☑ critical：表示如果服务在 4 分钟内存在多于 4 次，则系统重启到 recovery mode。
- ☑ disabled：表示服务不会自动启动，需要手动调用名字启动。
- ☑ setEnv <name> <value>：设置启动环境变量。
- ☑ socket <name> <type> <permission> [<user> [<group>]]：开启一个 unix 域的 socket，名字为 /dev/socket/<name>，<type>只能是 dgram 或者 stream，<user>和<group>默认为 0。
- ☑ user <username>：表示将用户切换为<username>，用户名已经定义好了，只能是 system/root。
- ☑ group <groupname>：表示将组切换为<groupname>。
- ☑ oneshot：表示这个 Service 只启动一次。
- ☑ class <name>：指定一个要启动的类，这个类中如果有多个 service，将会被同时启动。默认的 class 将会是 default。
- ☑ onrestart：在重启时执行一条命令。

（4）trigger

trigger 主要包括如下选项。

- ☑ boot：当/init.conf 加载完毕时。
- ☑ <name>=<value>：当<name>被设置为<value>时。
- ☑ device-added-<path>：设备<path>被添加时。
- ☑ device-removed-<path>：设备<path>被移除时。
- ☑ service-exited-<name>：服务<name>退出时。

（5）主要包含的操作命令及具体说明。

- ☑ exec <path> [<argument>]*：执行一个<path>指定的程序。
- ☑ export <name> <value>：设置一个全局变量。
- ☑ ifup <interface>：使网络接口<interface>连接。
- ☑ import <filename>：引入其他的配置文件。
- ☑ hostname <name>：设置主机名。
- ☑ chdir <directory>：切换工作目录。
- ☑ chmod <octal-mode> <path>：设置访问权限。
- ☑ chown <owner> <group> <path>：设置用户和组。
- ☑ chroot <directory>：设置根目录。
- ☑ class_start <serviceclass>：启动类中的 service。
- ☑ class_stop <serviceclass>：停止类中的 service。
- ☑ domainname <name>：设置域名。
- ☑ insmod <path>：安装模块。
- ☑ mkdir <path> [mode] [owner] [group]：创建一个目录，并可以指定权限、用户和组。
- ☑ mount <type> <device> <dir> [<mountoption>]*：加载指定设备到目录下。

☑ <mountoption>：包括 ro、rw、remount 和 noatime。
☑ setprop <name> <value>：设置系统属性。
☑ setrlimit <resource> <cur> <max>：设置资源访问权限。
☑ start <service>：开启服务。
☑ stop <service>：停止服务。
☑ symlink <target> <path>：创建一个动态链接。
☑ sysclktz <mins_west_of_gmt>：设置系统时钟。
☑ trigger <event>：触发事件。
☑ write <path> <string> [<string>]*：向<path>路径的文件写入多个<string>。

读者可以进到 Android 的 shell，会看到根目录有一个 init.rc 文件。启动 Android 后，会将文件 init.rc 装载到内存。而修改文件 init.rc 的内容实际上只是修改内存中 init.rc 文件的内容。一旦重启 Android，文件 init.rc 的内容又会恢复到最初的装载。想彻底修改文件 init.rc 内容，唯一的方式是修改 Android 的 ROM 中的内核镜像（boot.img）。

6.3.2　init.rc 解析

文件 init.rc 是一个配置文件，内部有许多语言规则，所有语言会在函数 init_parse_config_file()中进行解析。当前面的主函数 main()读取完配置文件 init.rc 后，会调用函数 parse_config()进行解析。整个实现流程如下所示。

```
init_parse_config_file—>read_file—>parse_config
```

（1）函数 parse_config 和 init_parse_config_file 在文件/system/core/init/init_parser.c 中实现。函数 parse_config()和 init_parse_config_file()的具体实现代码如下所示。

```
static void parse_config(const char *fn, char *s)//s 为 init.rc 中字符串的内容
{
    struct parse_state state;
    char *args[INIT_PARSER_MAXARGS];
    int nargs;

    nargs = 0;
    state.filename = fn;
    state.line = 1;
    state.ptr = s;
    state.nexttoken = 0;
    state.parse_line = parse_line_no_op;
    for (;;) {
        switch (next_token(&state)) {
        case T_EOF:          //文件的结尾
            state.parse_line(&state, 0, 0);
            return;
        case T_NEWLINE://新的一行
            if (nargs) {
                int kw = lookup_keyword(args[0]); //读取 init.rc 返回关键字，例如 service，返回 K_service
                if (kw_is(kw, SECTION)) {          //查看关键字是否为 SECTION，只有 service 和 on 满足
```

```
                    state.parse_line(&state, 0, 0);
                    parse_new_section(&state, kw, nargs, args);
                } else {
                    state.parse_line(&state, nargs, args);//on 和 service 两个段下面的内容
                }
                nargs = 0;
            }
            break;
        case T_TEXT://文本内容
            if (nargs < INIT_PARSER_MAXARGS) {
                args[nargs++] = state.text;
            }
            break;
        }
    }
}

int init_parse_config_file(const char *fn)
{
    char *data;
    data = read_file(fn, 0);
    if (!data) return -1;

    parse_config(fn, data);
    DUMP();
    return 0;
}
```

通过上述代码可以看出，在 for 的无限循环中对文件 init.rc 的内容进行了解析，以一行一行的形式进行了读取。

（2）每读取完一行内容换行到下一行时，使用函数 lookup_keyword()分析已经读取的一行的第一个参数。函数 lookup_keyword()的具体实现代码如下所示。

```
int lookup_keyword(const char *s)
{
    switch (*s++) {
    case 'c':
    if (!strcmp(s, "opy")) return K_copy;
        if (!strcmp(s, "apability")) return K_capability;
        if (!strcmp(s, "hdir")) return K_chdir;
        if (!strcmp(s, "hroot")) return K_chroot;
        if (!strcmp(s, "lass")) return K_class;
        if (!strcmp(s, "lass_start")) return K_class_start;
        if (!strcmp(s, "lass_stop")) return K_class_stop;
        if (!strcmp(s, "lass_reset")) return K_class_reset;
        if (!strcmp(s, "onsole")) return K_console;
        if (!strcmp(s, "hown")) return K_chown;
        if (!strcmp(s, "hmod")) return K_chmod;
        if (!strcmp(s, "ritical")) return K_critical;
        break;
```

```
    case 'd':
        if (!strcmp(s, "isabled")) return K_disabled;
        if (!strcmp(s, "omainname")) return K_domainname;
        break;
    case 'e':
        if (!strcmp(s, "xec")) return K_exec;
        if (!strcmp(s, "xport")) return K_export;
        break;
    case 'g':
        if (!strcmp(s, "roup")) return K_group;
        break;
    case 'h':
        if (!strcmp(s, "ostname")) return K_hostname;
        break;
    case 'i':
        if (!strcmp(s, "oprio")) return K_ioprio;
        if (!strcmp(s, "fup")) return K_ifup;
        if (!strcmp(s, "nsmod")) return K_insmod;
        if (!strcmp(s, "mport")) return K_import;
        break;
    case 'k':
        if (!strcmp(s, "eycodes")) return K_keycodes;
        break;
    case 'l':
        if (!strcmp(s, "oglevel")) return K_loglevel;
        if (!strcmp(s, "oad_persist_props")) return K_load_persist_props;
        break;
    case 'm':
        if (!strcmp(s, "kdir")) return K_mkdir;
        if (!strcmp(s, "ount_all")) return K_mount_all;
        if (!strcmp(s, "ount")) return K_mount;
        break;
    case 'o':
        if (!strcmp(s, "n")) return K_on;
        if (!strcmp(s, "neshot")) return K_oneshot;
        if (!strcmp(s, "nrestart")) return K_onrestart;
        break;
    case 'r':
        if (!strcmp(s, "estart")) return K_restart;
        if (!strcmp(s, "estorecon")) return K_restorecon;
        if (!strcmp(s, "mdir")) return K_rmdir;
        if (!strcmp(s, "m")) return K_rm;
        break;
    case 's':
        if (!strcmp(s, "eclabel")) return K_seclabel;
        if (!strcmp(s, "ervice")) return K_service;
        if (!strcmp(s, "etcon")) return K_setcon;
        if (!strcmp(s, "etenforce")) return K_setenforce;
        if (!strcmp(s, "etenv")) return K_setenv;
        if (!strcmp(s, "etkey")) return K_setkey;
```

```
            if (!strcmp(s, "etprop")) return K_setprop;
            if (!strcmp(s, "etrlimit")) return K_setrlimit;
            if (!strcmp(s, "etsebool")) return K_setsebool;
            if (!strcmp(s, "ocket")) return K_socket;
            if (!strcmp(s, "tart")) return K_start;
            if (!strcmp(s, "top")) return K_stop;
            if (!strcmp(s, "ymlink")) return K_symlink;
            if (!strcmp(s, "ysclktz")) return K_sysclktz;
            break;
        case 't':
            if (!strcmp(s, "rigger")) return K_trigger;
            break;
        case 'u':
            if (!strcmp(s, "ser")) return K_user;
            break;
        case 'w':
            if (!strcmp(s, "rite")) return K_write;
            if (!strcmp(s, "ait")) return K_wait;
            break;
        }
        return K_UNKNOWN;
}
```

由此可见，函数 lookup_keyword()主要对每一行的第一个字符做 case 判断，然后在 if 语句中调用 strcmp 命令，这些命令都是按照文件 init.rc 的格式要求进行的。例如，常用的 service 和 on 等经过 lookup_keyword 后返回 K_servcie 和 K_on。随后使用 kw_is(kw, SECTION)判断返回的 kw 是不是属于 Section 类型。在文件 init.rc 中，只有 service 和 on 满足该类型。

（3）定义关键字。在文件 keywords.h 中定义了 init 使用的关键字，在此文件中定义了如 do_class_start、do_class_stop 之类的函数，并且还定义了枚举。文件 keywords.h 的路径为/system/core/init/。

文件 keywords.h 的具体实现代码如下所示。

```
#ifndef KEYWORD
int do_chroot(int nargs, char **args);
int do_chdir(int nargs, char **args);
int do_class_start(int nargs, char **args);
int do_class_stop(int nargs, char **args);
int do_class_reset(int nargs, char **args);
…
#define __MAKE_KEYWORD_ENUM__
#define KEYWORD(symbol, flags, nargs, func) K_##symbol,
enum {
    K_UNKNOWN,
#endif
    KEYWORD(capability, OPTION,   0, 0)
    KEYWORD(chdir, COMMAND, 1, do_chdir)
    KEYWORD(chroot, COMMAND, 1, do_chroot)
    KEYWORD(class, OPTION,   0, 0)
    KEYWORD(class_start, COMMAND, 1, do_class_start)
```

```
    KEYWORD(class_stop, COMMAND, 1, do_class_stop)
    KEYWORD(class_reset, COMMAND, 1, do_class_reset)
    KEYWORD(console, OPTION,   0, 0)
    KEYWORD(critical, OPTION,   0, 0)
    KEYWORD(disabled, OPTION,   0, 0)
    KEYWORD(domainname, COMMAND, 1, do_domainname)
    KEYWORD(exec, COMMAND, 1, do_exec)
    KEYWORD(export, COMMAND, 2, do_export)
…
    KEYWORD(load_persist_props, COMMAND, 0, do_load_persist_props)
    KEYWORD(ioprio, OPTION,   0, 0)
#ifdef __MAKE_KEYWORD_ENUM__
    KEYWORD_COUNT,
};
#undef __MAKE_KEYWORD_ENUM__
#undef KEYWORD
#endif
```

文件 keywords.h 在文件 init_parser.c 中被用到了两次，具体引用代码如下所示。

```
#define SECTION 0x01
#define COMMAND 0x02
#define OPTION   0x04

#include "keywords.h"

#define KEYWORD(symbol, flags, nargs, func) \
    [ K_##symbol ] = { #symbol, func, nargs + 1, flags, },

struct {
    const char *name;
    int (*func)(int nargs, char **args);
    unsigned char nargs;
    unsigned char flags;
} keyword_info[KEYWORD_COUNT] = {
    [ K_UNKNOWN ] = { "unknown", 0, 0, 0 },
#include "keywords.h"
};
#undef KEYWORD

#define kw_is(kw, type) (keyword_info[kw].flags & (type))
#define kw_name(kw) (keyword_info[kw].name)
#define kw_func(kw) (keyword_info[kw].func)
#define kw_nargs(kw) (keyword_info[kw].nargs)
```

6.4 解析 Service

由前面的函数 lookup_keyword()可知，在调用过程中会对 on 和 service 所在的段进行解析，此处首

先分析 Service，在分析时以文件 init.rc 中的 service zygote 为例。

6.4.1　Zygote 对应的 service action

在文件 init.rc 中，Zygote 对应的 service action 的代码如下所示。

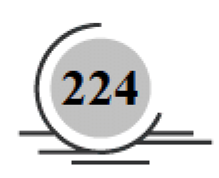
```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
    class main
    socket zygote stream 660 root system
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd
```

解析 action 的入口函数是 parse_new_section()，在此函数中再分别对 service 或者 on 关键字开头的内容进行解析。函数 parse_new_section()的具体实现代码如下所示。

```
void parse_new_section(struct parse_state *state, int kw,
                       int nargs, char **args)
{
    printf("[ %s %s ]\n", args[0],
           nargs > 1 ? args[1] : "");
    switch(kw) {
    case K_service:
        state->context = parse_service(state, nargs, args);
        if (state->context) {
            state->parse_line = parse_line_service;
            return;
        }
        break;
    case K_on:
        state->context = parse_action(state, nargs, args);
        if (state->context) {
            state->parse_line = parse_line_action;
            return;
        }
        break;
    case K_import:
        parse_import(state, nargs, args);
        break;
    }
    state->parse_line = parse_line_no_op;
}
```

6.4.2　init 组织 Service

在 init 进程中，使用了一个名为 service 的结构体保存和 service action 有关的信息。此结构体是在文件/system/core/init/init.h 中定义的。



结构体 service 的具体实现代码如下所示。

```
struct service {
        /* list of all services */
    struct listnode slist;

    const char *name;
    const char *classname;

    unsigned flags;
    pid_t pid;
    time_t time_started;    /* time of last start */
    time_t time_crashed;    /* first crash within inspection window */
    int nr_crashed;         /* number of times crashed within window */

    uid_t uid;
    gid_t gid;
    gid_t supp_gids[NR_SVC_SUPP_GIDS];
    size_t nr_supp_gids;

#ifdef HAVE_SELINUX
    char *seclabel;
#endif

    struct socketinfo *sockets;
    struct svcenvinfo *envvars;

    struct action onrestart;  /* Actions to execute on restart. */

    /* keycodes for triggering this service via /dev/keychord */
    int *keycodes;
    int nkeycodes;
    int keychord_id;

    int ioprio_class;
    int ioprio_pri;

    int nargs;
    /* "MUST BE AT THE END OF THE STRUCT" */
    char *args[1];
}; /* 'args' MUST be at the end of this struct */
```

另外，在文件 init.h 中还定义了结构体 action，具体实现代码如下所示。

```
struct action {
        /* node in list of all actions */
    struct listnode alist;
        /* node in the queue of pending actions */
    struct listnode qlist;
        /* node in list of actions for a trigger */
    struct listnode tlist;
```

```
	unsigned hash;
	const char *name;

	struct listnode commands;
	struct command *current;
};
```

这样便通过上述两个结构体对 Service 进行了组织。

6.4.3 解析 Service 用到的函数

在解析 Service 时会用到两个函数，分别是 parse_service()和 parse_line_service()。

（1）当解析文件 init.rc 中的 service zygote 时会执行函数 parse_service，此函数的功能是构建 Service 的骨架，对 Service 关键字开头的内容进行解析。函数 parse_service()的具体实现代码如下所示。

```
static void *parse_service(struct parse_state *state, int nargs, char **args)
{
	struct service *svc;
	if (nargs < 3) {
		parse_error(state, "services must have a name and a program\n");
		return 0;
	}
	if (!valid_name(args[1])) {
		parse_error(state, "invalid service name '%s'\n", args[1]);
		return 0;
	}

	svc = service_find_by_name(args[1]);//查找服务是否已经存在
	if (svc) {
		parse_error(state, "ignored duplicate definition of service '%s'\n", args[1]);
		return 0;
	}

	nargs -= 2;
	svc = calloc(1, sizeof(*svc) + sizeof(char*) * nargs);
	if (!svc) {
		parse_error(state, "out of memory\n");
		return 0;
	}
	svc->name = args[1];                //Service 的名字
	svc->classname = "default"; //svc 的类名默认是 default
	memcpy(svc->args, args + 2, sizeof(char*) * nargs);//首个参数存放的是可执行文件
	svc->args[nargs] = 0;
	svc->nargs = nargs;//参数个数
	svc->onrestart.name = "onrestart";
	list_init(&svc->onrestart.commands);
	list_add_tail(&service_list, &svc->slist);
	return svc;
}
```

在上述代码中，agrs[1]就是 Zygote，系统会先查找是否已经存在该服务，然后构建一个 service svc 并进行相关的填充，包括服务名、服务所属的类别名字和已经服务启动带入的参数个数（要减去 Service 和服务名 Zygote），最后将这个 svc 加入到 service_list 全局链表中。

（2）函数 parse_line_service()的功能是，根据配置文件的内容填充 Service 结构体，并解析 Service 中剩余行中的 Option，例如 class、socket、onrestart 等。函数 parse_line_service()的具体实现代码如下所示。

```
static void parse_line_service(struct parse_state *state, int nargs, char **args)
{
    struct service *svc = state->context;
    struct command *cmd;
    int i, kw, kw_nargs;

    if (nargs == 0) {
        return;
    }

    svc->ioprio_class = IoSchedClass_NONE;

    kw = lookup_keyword(args[0]);
    switch (kw) {
    case K_capability:
        break;
    case K_class:
        if (nargs != 2) {
            parse_error(state, "class option requires a classname\n");
        } else {
            svc->classname = args[1];
        }
        break;
    case K_console:
        svc->flags |= SVC_CONSOLE;
        break;
    case K_disabled:
        svc->flags |= SVC_DISABLED;
        svc->flags |= SVC_RC_DISABLED;
        break;
    case K_ioprio:
        if (nargs != 3) {
            parse_error(state, "ioprio optin usage: ioprio <rt|be|idle> <ioprio 0-7>\n");
        } else {
            svc->ioprio_pri = strtoul(args[2], 0, 8);

            if (svc->ioprio_pri < 0 || svc->ioprio_pri > 7) {
                parse_error(state, "priority value must be range 0 - 7\n");
                break;
            }

            if (!strcmp(args[1], "rt")) {
```

```
                svc->ioprio_class = IoSchedClass_RT;
            } else if (!strcmp(args[1], "be")) {
                svc->ioprio_class = IoSchedClass_BE;
            } else if (!strcmp(args[1], "idle")) {
                svc->ioprio_class = IoSchedClass_IDLE;
            } else {
                parse_error(state, "ioprio option usage: ioprio <rt|be|idle> <0-7>\n");
            }
        }
        break;
    case K_group:
        if (nargs < 2) {
            parse_error(state, "group option requires a group id\n");
        } else if (nargs > NR_SVC_SUPP_GIDS + 2) {
            parse_error(state, "group option accepts at most %d supp. groups\n",
                        NR_SVC_SUPP_GIDS);
        } else {
            int n;
            svc->gid = decode_uid(args[1]);
            for (n = 2; n < nargs; n++) {
                svc->supp_gids[n-2] = decode_uid(args[n]);
            }
            svc->nr_supp_gids = n - 2;
        }
        break;
    case K_keycodes:
        if (nargs < 2) {
            parse_error(state, "keycodes option requires atleast one keycode\n");
        } else {
            svc->keycodes = malloc((nargs - 1) * sizeof(svc->keycodes[0]));
            if (!svc->keycodes) {
                parse_error(state, "could not allocate keycodes\n");
            } else {
                svc->nkeycodes = nargs - 1;
                for (i = 1; i < nargs; i++) {
                    svc->keycodes[i - 1] = atoi(args[i]);
                }
            }
        }
        break;
    case K_oneshot:
        svc->flags |= SVC_ONESHOT;
        break;
    case K_onrestart:
        nargs--;
        args++;
        kw = lookup_keyword(args[0]);
        if (!kw_is(kw, COMMAND)) {
            parse_error(state, "invalid command '%s'\n", args[0]);
            break;
        }
```

```
        kw_nargs = kw_nargs(kw);
        if (nargs < kw_nargs) {
            parse_error(state, "%s requires %d %s\n", args[0], kw_nargs - 1,
                kw_nargs > 2 ? "arguments" : "argument");
            break;
        }

        cmd = malloc(sizeof(*cmd) + sizeof(char*) * nargs);
        cmd->func = kw_func(kw);
        cmd->nargs = nargs;
        memcpy(cmd->args, args, sizeof(char*) * nargs);
        list_add_tail(&svc->onrestart.commands, &cmd->clist);
        break;
    case K_critical:
        svc->flags |= SVC_CRITICAL;
        break;
    case K_setenv: { /* name value */
        struct svcenvinfo *ei;
        if (nargs < 2) {
            parse_error(state, "setenv option requires name and value arguments\n");
            break;
        }
        ei = calloc(1, sizeof(*ei));
        if (!ei) {
            parse_error(state, "out of memory\n");
            break;
        }
        ei->name = args[1];
        ei->value = args[2];
        ei->next = svc->envvars;
        svc->envvars = ei;
        break;
    }
    case K_socket: {/* name type perm [ uid gid ] */
        struct socketinfo *si;
        if (nargs < 4) {
            parse_error(state, "socket option requires name, type, perm arguments\n");
            break;
        }
        if (strcmp(args[2],"dgram") && strcmp(args[2],"stream")
                && strcmp(args[2],"seqpacket")) {
            parse_error(state, "socket type must be 'dgram', 'stream' or 'seqpacket'\n");
            break;
        }
        si = calloc(1, sizeof(*si));
        if (!si) {
            parse_error(state, "out of memory\n");
            break;
        }
        si->name = args[1];
        si->type = args[2];
```

```
            si->perm = strtoul(args[3], 0, 8);
            if (nargs > 4)
                si->uid = decode_uid(args[4]);
            if (nargs > 5)
                si->gid = decode_uid(args[5]);
            si->next = svc->sockets;
            svc->sockets = si;
            break;
        }
        case K_user:
            if (nargs != 2) {
                parse_error(state, "user option requires a user id\n");
            } else {
                svc->uid = decode_uid(args[1]);
            }
            break;
        case K_seclabel:
#ifdef HAVE_SELINUX
            if (nargs != 2) {
                parse_error(state, "seclabel option requires a label string\n");
            } else {
                svc->seclabel = args[1];
            }
#endif
            break;

        default:
            parse_error(state, "invalid option '%s'\n", args[0]);
        }
}
```

6.5 解析 on

本节将详细剖析 on 字段的内容，以 on boot 这个 Section 为例进行分析。希望读者认真理解，为本书后面知识的学习打下基础。

6.5.1 Zygote 对应的 on action

字段 on 的内容比较复杂，本书将以 on boot 这个 Section 为例进行分析。在文件 init.rc 中，Zygote 对应的 on boot action 的代码如下所示。

```
on boot
# basic network init
    ifup lo
    hostname localhost
    domainname localdomain
```

```
# set RLIMIT_NICE to allow priorities from 19 to -20
    setrlimit 13 40 40

# Memory management.  Basic kernel parameters, and allow the high
# level system server to be able to adjust the kernel OOM driver
# parameters to match how it is managing things.
    write /proc/sys/vm/overcommit_memory 1
    write /proc/sys/vm/min_free_order_shift 4
    chown root system /sys/module/lowmemorykiller/parameters/adj
    chmod 0664 /sys/module/lowmemorykiller/parameters/adj
    chown root system /sys/module/lowmemorykiller/parameters/minfree
    chmod 0664 /sys/module/lowmemorykiller/parameters/minfree

    # Tweak background writeout
    write /proc/sys/vm/dirty_expire_centisecs 200
    write /proc/sys/vm/dirty_background_ratio  5

    # Permissions for System Server and daemons
    chown radio system /sys/android_power/state
    chown radio system /sys/android_power/request_state
    chown radio system /sys/android_power/acquire_full_wake_lock
    chown radio system /sys/android_power/acquire_partial_wake_lock
    chown radio system /sys/android_power/release_wake_lock
    chown system system /sys/power/autosleep
    chown system system /sys/power/state
    chown system system /sys/power/wakeup_count
    chown radio system /sys/power/wake_lock
    chown radio system /sys/power/wake_unlock
    chmod 0660 /sys/power/state
    chmod 0660 /sys/power/wake_lock
    chmod 0660 /sys/power/wake_unlock
…
# Set this property so surfaceflinger is not started by system_init
    setprop system_init.startsurfaceflinger 0

    class_start core
    class_start main
```

与前面对 Service 的分析类似，case 中进入 K_on 选项执行函数 parse_action()。函数 parse_action()的具体实现代码如下所示。

```
static void *parse_action(struct parse_state *state, int nargs, char **args)
{
    struct action *act;
    if (nargs < 2) {
        parse_error(state, "actions must have a trigger\n");
        return 0;
    }
    if (nargs > 2) {
        parse_error(state, "actions may not have extra parameters\n");
```

```
            return 0;
        }
        act = calloc(1, sizeof(*act));
        act->name = args[1];
        list_init(&act->commands);
        list_add_tail(&action_list, &act->alist);
            /* XXX add to hash */
        return act;
}
```

6.5.2 结构体 action

在 init 进程中可以看到一个名为 action 结构体类似于 service，这个 action 的名字为 boot，最后会将这个 action 加入到全局链表 action_list 中。结构体 action 的具体实现代码如下所示。

```
struct action {
        /* node in list of all actions */
    struct listnode alist;
        /* node in the queue of pending actions */
    struct listnode qlist;
        /* node in list of actions for a trigger */
    struct listnode tlist;

    unsigned hash;
    const char *name;

    struct listnode commands;
    struct command *current;
};
```

6.5.3 解析 on 字段所在的 option

在解析 on 时会用到函数 parse_line_action()，功能是对 on 字段所在的 option 进行解析。函数 parse_line_action()的具体实现代码如下所示。

```
static void parse_line_action(struct parse_state* state, int nargs, char **args) //action 所在的行
{
    struct command *cmd;
    struct action *act = state->context;//on boot 启动
    int (*func)(int nargs, char **args);
    int kw, n;

    if (nargs == 0) {
        return;
    }

    kw = lookup_keyword(args[0]);//命令的参数个数
    if (!kw_is(kw, COMMAND)) {
```

```
        parse_error(state, "invalid command '%s'\n", args[0]);
        return;
    }

    n = kw_nargs(kw);
    if (nargs < n) {
        parse_error(state, "%s requires %d %s\n", args[0], n - 1,
            n > 2 ? "arguments" : "argument");
        return;
    }
    cmd = malloc(sizeof(*cmd) + sizeof(char*) * nargs);
    cmd->func = kw_func(kw);
    cmd->nargs = nargs;
    memcpy(cmd->args, args, sizeof(char*) * nargs);
    list_add_tail(&act->commands, &cmd->clist);     //
}
```

到此为止，on 和 service 两个 Section 的分析工作全部完成。

6.6 init 控制 Service

在 Android 5.0 系统中，当进行 init 进程初始化时，除了对系统做一些必要的初始化外操作，还需要启动 Service 进程，而 Service 是在 init 脚本中定义的。本节将详细讲解在 init 中控制 Service 的知识。

6.6.1 启动 Zygote

Android 系统是基于 Linux 内核的，而在 Linux 系统中的所有进程都是 init 进程的子进程或孙进程。也就是说，所有的进程都是直接或者间接地由 init 进程 fork 出来的。Zygote 进程也不例外，它是在系统启动的过程中，由 init 进程创建的。在系统启动脚本文件 system/core/rootdir/init.rc 中，可以看到如下启动 Zygote 进程的脚本命令：

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
    socket zygote stream 666
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd
```

在上述代码中，各个关键字的具体说明如下所示。

☑ service：用于通知 init 进程创建一个名为 Zygote 的进程，这个 Zygote 进程要执行的程序是/system/bin/app_process，后面是要传给 app_process 的参数。

☑ Socket：表示这个 Zygote 进程需要一个名称为 Zygote 的 Socket 资源，这样启动系统后，就可以在/dev/socket 目录下看到有一个名为 Zygote 的文件。这里定义的 Socket 的类型为 unix domain socket，是用来作本地进程间通信用的。

6.6.2 启动 Service

首先看函数 do_class_start()，此函数的功能是启动 Service，此函数在文件/system/core/init/builtins.c 中定义。

函数 do_class_start()的具体实现代码如下所示。

```
int do_class_start(int nargs, char **args)
{
            /* Starting a class does not start services
             * which are explicitly disabled.   They must
             * be started individually
             */
        service_for_each_class(args[1], service_start_if_not_disabled);
        return 0;
}
```

在上述代码中，调用了函数 service_start_if_not_disabled()实现启动功能，此函数也在文件 builtins.c 中实现，具体实现代码如下所示。

```
static void service_start_if_not_disabled(struct service *svc)
{
        if (!(svc->flags & SVC_DISABLED)) {
              service_start(svc, NULL);
        }
}
```

在上述代码中，调用了函数 service_start()实现启动功能。函数 service_start()是整个启动功能的核心，在文件 init.c 中定义，具体实现代码如下所示。

```
void service_start(struct service *svc, const char *dynamic_args)
{
        struct stat s;
        pid_t pid;
        int needs_console;
        int n;
#ifdef HAVE_SELINUX
        char *scon = NULL;
        int rc;
#endif
            /* starting a service removes it from the disabled or reset
             * state and immediately takes it out of the restarting
             * state if it was in there
             */
        svc->flags &= (~(SVC_DISABLED|SVC_RESTARTING|SVC_RESET));
        svc->time_started = 0;
        if (svc->flags & SVC_RUNNING) {
              return;
        }
```

```
    needs_console = (svc->flags & SVC_CONSOLE) ? 1 : 0;
    if (needs_console && (!have_console)) {
        ERROR("service '%s' requires console\n", svc->name);
        svc->flags |= SVC_DISABLED;
        return;
    }

    if (stat(svc->args[0], &s) != 0) {
        ERROR("cannot find '%s', disabling '%s'\n", svc->args[0], svc->name);
        svc->flags |= SVC_DISABLED;
        return;
    }

    if ((!(svc->flags & SVC_ONESHOT)) && dynamic_args) {
        ERROR("service '%s' must be one-shot to use dynamic args, disabling\n",
                svc->args[0]);
        svc->flags |= SVC_DISABLED;
        return;
    }

#ifdef HAVE_SELINUX
    if (is_selinux_enabled() > 0) {
        char *mycon = NULL, *fcon = NULL;

        INFO("computing context for service '%s'\n", svc->args[0]);
        rc = getcon(&mycon);
        if (rc < 0) {
            ERROR("could not get context while starting '%s'\n", svc->name);
            return;
        }

        rc = getfilecon(svc->args[0], &fcon);
        if (rc < 0) {
            ERROR("could not get context while starting '%s'\n", svc->name);
            freecon(mycon);
            return;
        }

        rc = security_compute_create(mycon, fcon, string_to_security_class("process"), &scon);
        freecon(mycon);
        freecon(fcon);
        if (rc < 0) {
            ERROR("could not get context while starting '%s'\n", svc->name);
            return;
        }
    }
#endif

    NOTICE("starting '%s'\n", svc->name);

    pid = fork();
```

```
    if (pid == 0) {
        struct socketinfo *si;
        struct svcenvinfo *ei;
        char tmp[32];
        int fd, sz;

        umask(077);
#ifdef __arm__
        int current = personality(0xffffFFFF);
        personality(current | ADDR_COMPAT_LAYOUT);
#endif
        if (properties_inited()) {
            get_property_workspace(&fd, &sz);
            sprintf(tmp, "%d,%d", dup(fd), sz);
            add_environment("ANDROID_PROPERTY_WORKSPACE", tmp);
        }

        for (ei = svc->envvars; ei; ei = ei->next)
            add_environment(ei->name, ei->value);

#ifdef HAVE_SELINUX
        setsockcreatecon(scon);
#endif

        for (si = svc->sockets; si; si = si->next) {
            int socket_type = (
                    !strcmp(si->type, "stream") ? SOCK_STREAM :
                        (!strcmp(si->type, "dgram") ? SOCK_DGRAM : SOCK_SEQPACKET));
            int s = create_socket(si->name, socket_type,
                                  si->perm, si->uid, si->gid);
            if (s >= 0) {
                publish_socket(si->name, s);
            }
        }

#ifdef HAVE_SELINUX
        freecon(scon);
        scon = NULL;
        setsockcreatecon(NULL);
#endif

        if (svc->ioprio_class != IoSchedClass_NONE) {
            if (android_set_ioprio(getpid(), svc->ioprio_class, svc->ioprio_pri)) {
                ERROR("Failed to set pid %d ioprio = %d,%d: %s\n",
                      getpid(), svc->ioprio_class, svc->ioprio_pri, strerror(errno));
            }
        }

        if (needs_console) {
            setsid();
```

```
            open_console();
        } else {
            zap_stdio();
        }

#if 0
        for (n = 0; svc->args[n]; n++) {
            INFO("args[%d] = '%s'\n", n, svc->args[n]);
        }
        for (n = 0; ENV[n]; n++) {
            INFO("env[%d] = '%s'\n", n, ENV[n]);
        }
#endif

        setpgid(0, getpid());

    /* as requested, set our gid, supplemental gids, and uid */
        if (svc->gid) {
            if (setgid(svc->gid) != 0) {
                ERROR("setgid failed: %s\n", strerror(errno));
                _exit(127);
            }
        }
        if (svc->nr_supp_gids) {
            if (setgroups(svc->nr_supp_gids, svc->supp_gids) != 0) {
                ERROR("setgroups failed: %s\n", strerror(errno));
                _exit(127);
            }
        }
        if (svc->uid) {
            if (setuid(svc->uid) != 0) {
                ERROR("setuid failed: %s\n", strerror(errno));
                _exit(127);
            }
        }

#ifdef HAVE_SELINUX
        if (svc->seclabel) {
            if (is_selinux_enabled() > 0 && setexeccon(svc->seclabel) < 0) {
                ERROR("cannot setexeccon('%s'): %s\n", svc->seclabel, strerror(errno));
                _exit(127);
            }
        }
#endif

        if (!dynamic_args) {
            if (execve(svc->args[0], (char**) svc->args, (char**) ENV) < 0) {
                ERROR("cannot execve('%s'): %s\n", svc->args[0], strerror(errno));
            }
        } else {
            char *arg_ptrs[INIT_PARSER_MAXARGS+1];
```

```
            int arg_idx = svc->nargs;
            char *tmp = strdup(dynamic_args);
            char *next = tmp;
            char *bword;

            /* Copy the static arguments */
            memcpy(arg_ptrs, svc->args, (svc->nargs * sizeof(char *)));

            while((bword = strsep(&next, " "))) {
                arg_ptrs[arg_idx++] = bword;
                if (arg_idx == INIT_PARSER_MAXARGS)
                    break;
            }
            arg_ptrs[arg_idx] = '\0';
            execve(svc->args[0], (char**) arg_ptrs, (char**) ENV);
        }
        _exit(127);
    }

#ifdef HAVE_SELINUX
    freecon(scon);
#endif

    if (pid < 0) {
        ERROR("failed to start '%s'\n", svc->name);
        svc->pid = 0;
        return;
    }

    svc->time_started = gettime();
    svc->pid = pid;
    svc->flags |= SVC_RUNNING;

    if (properties_inited())
        notify_service_state(svc->name, "running");
}
```

在函数 service_start()中，参数 dynamic_args 只有当 Service 的 option 中有 oneshot 时才会用到，此时会通过替换掉启动服务的命令参数启动服务。因为 Service 的 option 会记录在 struct service 中，所以在启动 Service 时只需考虑到这些选项即可。同时，会记录下 Service 的 pid 和状态等信息。

6.6.3 总结 4 种启动 Service 的方式

其实在 init 进程中，可以使用如下方式启动 Service。

（1）在 action 下面添加和启动服务相关的 command，在 action 中和操作服务相关的命令如下。

☑ class_start <serviceclass> #：启动所有指定 class 的服务。

☑ class_stop <serviceclass> #：停止所有指定 class 的服务，后续无法通过 class_start 启动。

☑ class_reset <serviceclass> #：停止服务，后续可以通过 class_start 启动。

☑ restart <servicename> #：重启指定名称的服务，先 stop，再 start。

☑ start <servicename> #：启动指定名称的服务。

☑ stop <servicename> #：停止指定名称的服务。

在启动 command 时，在文件 init.c 的主函数 main()中，通过使用 for(;;)循环执行函数 execute_one_command()的方式实现，此函数的具体实现代码如下所示。

```
void execute_one_command(void)
{
    int ret;

    if (!cur_action || !cur_command || is_last_command(cur_action, cur_command)) {
        cur_action = action_remove_queue_head();
        cur_command = NULL;
        if (!cur_action)
            return;
        INFO("processing action %p (%s)\n", cur_action, cur_action->name);
        cur_command = get_first_command(cur_action);
    } else {
        cur_command = get_next_command(cur_action, cur_command);
    }

    if (!cur_command)
        return;

    ret = cur_command->func(cur_command->nargs, cur_command->args);//执行 class_start 等
    INFO("command '%s' r=%d\n", cur_command->args[0], ret);
}
```

（2）使用函数 restart_processes()和 restart_service_if_needed()重启 Service，该函数位于 init 的主线程循环中，功能是查看是否有需要重新启动的 service。在文件 init.c 中，函数 restart_processes()和 restart_service_if_needed()的具体实现代码如下所示。

```
static void restart_service_if_needed(struct service *svc)
{
    time_t next_start_time = svc->time_started + 5;

    if (next_start_time <= gettime()) {
        svc->flags &= (~SVC_RESTARTING);
        service_start(svc, NULL);
        return;
    }

    if ((next_start_time < process_needs_restart) ||
        (process_needs_restart == 0)) {
        process_needs_restart = next_start_time;
    }
}
```

```
static void restart_processes()
{
    process_needs_restart = 0;
    service_for_each_flags(SVC_RESTARTING,
                           restart_service_if_needed);
}
```

在重启过程中，会重启 flag 为 SVC_RESTARTING 的服务。这部分进程的重启其实在 init 由 handle_signal 来管理，一旦出现 Service 崩溃，函数 poll()会接收到相关文件变化的信息，并执行 handle_signal 中的函数 wait_for_one_process()。函数 wait_for_one_process()的具体实现代码如下所示。

```
static int wait_for_one_process(int block)
{
    pid_t pid;
    int status;
    struct service *svc;
    struct socketinfo *si;
    time_t now;
    struct listnode *node;
    struct command *cmd;

    while ( (pid = waitpid(-1, &status, block ? 0 : WNOHANG)) == -1 && errno == EINTR );
    if (pid <= 0) return -1;
    INFO("waitpid returned pid %d, status = %08x\n", pid, status);

    svc = service_find_by_pid(pid);
    if (!svc) {
        ERROR("untracked pid %d exited\n", pid);
        return 0;
    }
…
  svc->flags |= SVC_RESTARTING;

    /* Execute all onrestart commands for this service */
    list_for_each(node, &svc->onrestart.commands) {
        cmd = node_to_item(node, struct command, clist);
        cmd->func(cmd->nargs, cmd->args);
    }
    notify_service_state(svc->name, "restarting");
    return 0;
}
```

在上述代码中，使用 waitpid 找到子进程退出的进程号 pid，然后查找到该 Service，对 Service 中的 onrestart 这个 commands 进行操作。同时将 Service 的 flag 设置为 SVC_RESTARTING，这样就结合前面讲到的 restart_processes 重新启动了该服务进程。

（3）在文件/system/core/init/property_service.c 中，使用函数 handle_property_set_f()向 Socket 中名称为 property_service 的属性服务发送控制的消息，这样便可以进入到该函数中。函数 handle_property_set_f()的具体实现代码如下所示。



```
void handle_property_set_fd()
{
    prop_msg msg;
    int s;
    int r;
    int res;
    struct ucred cr;
    struct sockaddr_un addr;
    socklen_t addr_size = sizeof(addr);
    socklen_t cr_size = sizeof(cr);
    char * source_ctx = NULL;

    if ((s = accept(property_set_fd, (struct sockaddr *) &addr, &addr_size)) < 0) {
        return;
    }

    /* Check socket options here */
    if (getsockopt(s, SOL_SOCKET, SO_PEERCRED, &cr, &cr_size) < 0) {
        close(s);
        ERROR("Unable to recieve socket options\n");
        return;
    }

    r = TEMP_FAILURE_RETRY(recv(s, &msg, sizeof(msg), 0));
    if(r != sizeof(prop_msg)) {
        ERROR("sys_prop: mis-match msg size recieved: %d expected: %d errno: %d\n",
              r, sizeof(prop_msg), errno);
        close(s);
        return;
    }

    switch(msg.cmd) {
    case PROP_MSG_SETPROP:
        msg.name[PROP_NAME_MAX-1] = 0;
        msg.value[PROP_VALUE_MAX-1] = 0;

#ifdef HAVE_SELINUX
        getpeercon(s, &source_ctx);
#endif

        if(memcmp(msg.name,"ctl.",4) == 0) {
            close(s);
            if (check_control_perms(msg.value, cr.uid, cr.gid, source_ctx)) {
                handle_control_message((char*) msg.name + 4, (char*) msg.value);
            } else {
                ERROR("sys_prop: Unable to %s service ctl [%s] uid:%d gid:%d pid:%d\n",
                        msg.name + 4, msg.value, cr.uid, cr.gid, cr.pid);
            }
        } else {
```

```
            if (check_perms(msg.name, cr.uid, cr.gid, source_ctx)) {
                property_set((char*) msg.name, (char*) msg.value);
            } else {
                ERROR("sys_prop: permission denied uid:%d  name:%s\n",
                      cr.uid, msg.name);
            }
            close(s);
        }
#ifdef HAVE_SELINUX
        freecon(source_ctx);
#endif

        break;

    default:
        close(s);
        break;
    }
}
```

（4）使用函数 handle_keychord()启动，该函数和 chorded keyboard 有关，功能是处理注册在 Service structure 上的 keychord，通常是启动 Service。函数 handle_keychord()在文件 system/core/init/keychords.c 中定义，具体实现代码如下所示。

```
void handle_keychord()
{
    struct service *svc;
    const char* debuggable;
    const char* adb_enabled;
    int ret;
    __u16 id;

    debuggable = property_get("ro.debuggable");
    adb_enabled = property_get("init.svc.adbd");
    ret = read(keychord_fd, &id, sizeof(id));
    if (ret != sizeof(id)) {
        ERROR("could not read keychord id\n");
        return;
    }

    if ((debuggable && !strcmp(debuggable, "1")) ||
        (adb_enabled && !strcmp(adb_enabled, "running"))) {
        svc = service_find_by_keychord(id);
        if (svc) {
            INFO("starting service %s from keychord\n", svc->name);
            service_start(svc, NULL);
        } else {
            ERROR("service for keychord %d not found\n", id);
        }
    }
}
```

6.7　启动属性服务

init 在启动的过程中会启动属性服务（Socket 服务），并且在内存中建立一块存储这些属性的存储区域。当读取这些属性时，直接从这一内存区域读取，如果修改属性值，需要通过 Socket 连接属性服务完成。在文件 init.c 中，函数 action()调用函数 start_property_service()来启动属性服务，action 是 init.rc 及其类似文件中的一种执行机制。

在文件 init.c 中，可以看到和属性操作相关的代码情景，例如：

```
property_init()
property_set_fd
```

本节将详细讲解在 Android 5.0 系统中 init 控制属性服务的基本知识。

6.7.1　引入属性

在文件 init.c 的主函数 main()中，调用函数 property_init()为属性分配了一些存储空间。如果查看文件 init.rc，会发现该文件开始部分用一些 import 语句导入了其他配置文件，例如，/init.usb.rc。大多数配置文件都直接使用了确定的文件名，只有如下的代码使用了一个变量（${ro.hardware}）执行了配置文件名的一部分。

```
import /init.${ro.hardware}.rc
```

要想了解上述变量的获得过程，首先需要了解配置文件 init.${ro.hardware}.rc 的内容，这些通常与当前的硬件有关，其中，函数 get_hardware_name()用于获取硬件的名称信息，具体代码如下所示。

```
void get_hardware_name(char *hardware, unsigned int *revision)
{
    char data[1024];
    int fd, n;
    char *x, *hw, *rev;

    /*如果 hardware 已经有值了，说明 hardware 通过内核命令行提供，直接返回*/
    if (hardware[0])
        return;
    //打开/proc/cpuinfo 文件
    fd = open("/proc/cpuinfo", O_RDONLY);
    if (fd < 0) return;
    //读取/proc/cpuinfo 文件的内容
    n = read(fd, data, 1023);
    close(fd);
    if (n < 0) return;

    data[n] = 0;
    //从/proc/cpuinfo 文件中获取 Hardware 字段的值
    hw = strstr(data, "\nHardware");
```

```
        rev = strstr(data, "\nRevision");
        //成功获取 Hardware 字段的值
        if (hw) {
            x = strstr(hw, ": ");
            if (x) {
                x += 2;
                n = 0;
                while (*x && *x != '\n') {
                    if (!isspace(*x))
                        //将 Hardware 字段的值都转换为小写，并更新 Hardware 参数的值
                        //Hardware 也就是在 init.c 文件中定义的 Hardware 数组
                        hardware[n++] = tolower(*x);
                    x++;
                    if (n == 31) break;
                }
                hardware[n] = 0;
            }
        }

        if (rev) {
            x = strstr(rev, ": ");
            if (x) {
                *revision = strtoul(x + 2, 0, 16);
            }
        }
}
```

从上述代码可以得知，该函数主要用于确定 hardware 和 revision 变量的值。获取 hardware 的来源是从 Linux 内核命令行或文件/proc/cpuinfo 中的内容，文件/proc/cpuinfo 是虚拟文件（内存文件），执行 cat /proc/cpuinfo 命令会看到该文件中的内容，如图 6-1 所示。

```
Processor       : ARMv7 Processor rev 9 (v7l)
processor       : 0
BogoMIPS        : 1993.93

processor       : 1
BogoMIPS        : 1993.93

processor       : 2
BogoMIPS        : 1993.93

processor       : 3
BogoMIPS        : 1993.93

Features        : swp half thumb fastmult vfp edsp neon vfpv3 tls
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x2
CPU part        : 0xc09
CPU revision    : 9

Hardware        : grouper
Revision        : 0000
Serial          : 0f410a0001440200
root@android:/ #
```

图 6-1　显示文件内容

在图 6-1 中，白框中的内容就是 Hardware 字段的值。由于该设备是 Nexus 7，所以值为 grouper。如果程序就到此位置，那么与硬件有关的配置文件名是 init.grouper.rc。有 Nexus 7 的读者会看到在根目录下确实有一个 init.grouper.rc 文件。说明 Nexus 7 的原生 ROM 并没有在其他的地方设置配置文件名，所以配置文件名就是从/proc/cpuinfo 文件的 Hardware 字段中获取的值。



6.7.2　设置内核变量

接下来看在函数 get_hardware_name()后面调用的函数 process_kernel_cmdline()，具体实现代码如下所示。

```
static void process_kernel_cmdline(void)
{
	/* don't expose the raw commandline to nonpriv processes */
	chmod("/proc/cmdline", 0440);

	//导入内核命令行参数
	import_kernel_cmdline(0, import_kernel_nv);
	if (qemu[0])
		import_kernel_cmdline(1, import_kernel_nv);

	//用属性值设置内核变量
	export_kernel_boot_props();
}
```

在上述代码中，除了使用函数 import_kernel_cmdline()导入内核变量外，其主要功能是调用函数 export_kernel_boot_props()通过属性设置内核变量。例如，通过属性 ro.boot.hardware 设置 hardware 变量。也就是说，可以通过属性值 ro.boot.hardware 修改函数 get_hardware_name()从文件/proc/cpuinfo 中得到的 hardware 字段值。函数 export_kernel_boot_props()的具体实现代码如下所示。

```
static void export_kernel_boot_props(void)
{
	char tmp[PROP_VALUE_MAX];
	const char *pval;
	unsigned i;
	struct {
		const char *src_prop;
		const char *dest_prop;
		const char *def_val;
	} prop_map[] = {
		{ "ro.boot.serialno", "ro.serialno", "", },
		{ "ro.boot.mode", "ro.bootmode", "unknown", },
		{ "ro.boot.baseband", "ro.baseband", "unknown", },
		{ "ro.boot.bootloader", "ro.bootloader", "unknown", },
	};
	//通过内核的属性设置应用层配置文件的属性
	for (i = 0; i < ARRAY_SIZE(prop_map); i++) {
		pval = property_get(prop_map[i].src_prop);
		property_set(prop_map[i].dest_prop, pval ?: prop_map[i].def_val);
	}
	//根据 ro.boot.console 属性的值设置 console 变量
	pval = property_get("ro.boot.console");
	if (pval)
		strlcpy(console, pval, sizeof(console));
```

```
    /* save a copy for init's usage during boot */
    strlcpy(bootmode, property_get("ro.bootmode"), sizeof(bootmode));

    /* if this was given on kernel command line, override what we read
     * before (e.g. from /proc/cpuinfo), if anything */
    //获取 ro.boot.hardware 属性的值
    pval = property_get("ro.boot.hardware");
    if (pval)
        //这里通过 ro.boot.hardware 属性再次改变 hardware 变量的值
        strlcpy(hardware, pval, sizeof(hardware));
    //利用 hardware 变量的值设置 ro.hardware 属性
    //这个属性就是前面提到的设置初始化文件名的属性，实际上是通过 hardware 变量设置的
    property_set("ro.hardware", hardware);

    snprintf(tmp, PROP_VALUE_MAX, "%d", revision);
    property_set("ro.revision", tmp);

    /* TODO: these are obsolete. We should delete them */
    if (!strcmp(bootmode,"factory"))
        property_set("ro.factorytest", "1");
    else if (!strcmp(bootmode,"factory2"))
        property_set("ro.factorytest", "2");
    else
        property_set("ro.factorytest", "0");
}
```

由上述代码可以看出，该函数实际上就是来回设置一些属性值，并且利用某些属性值修改 console、hardware 等变量。其中 hardware 变量（就是一个长度为 32 的字符数组）在函数 get_hardware_name()中已经从文件/proc/cpuinfo 中获得过一次值了，在函数 export_kernel_boot_props()中又通过属性 ro.boot.hardware 设置了一次值，不过在 Nexus 7 中并没有设置该属性，所以 hardware 的值仍为 grouper。最后用变量 hardware 设置属性 ro.hardware，所以最后的初始化文件名为 init.grouper.rc。

6.7.3 初始化属性服务

在文件/system/core/init/property_service.c 中，使用函数 property_init()初始化属性存储区域，具体实现代码如下所示。

```
void property_init(void)
{
    init_property_area();
}
```

在上述代码中，函数 init_property_area()也是在文件 property_service.c 中实现的，该函数用于初始化属性内存区域，也就是__system_property_area__变量。函数 init_property_area()的具体实现代码如下所示。

```
static int init_property_area(void)
{
```



```
    prop_area *pa;

    if(pa_info_array)
        return -1;

    if(init_workspace(&pa_workspace, PA_SIZE))
        return -1;

    fcntl(pa_workspace.fd, F_SETFD, FD_CLOEXEC);

    pa_info_array = (void*) (((char*) pa_workspace.data) + PA_INFO_START);

    pa = pa_workspace.data;
    memset(pa, 0, PA_SIZE);
    pa->magic = PROP_AREA_MAGIC;
    pa->version = PROP_AREA_VERSION;

        /* plug into the lib property services */
    __system_property_area__ = pa;
    property_area_inited = 1;
    return 0;
}
```

6.7.4　实现具体启动工作

在文件/system/core/init/property_service.c 中，使用函数 start_property_service()启动一个属性服务器，具体实现代码如下所示。

```
void start_property_service(void)
{
    int fd;
    //装载不同的属性文件
    load_properties_from_file(PROP_PATH_SYSTEM_BUILD);
    load_properties_from_file(PROP_PATH_SYSTEM_DEFAULT);
    load_override_properties();
    /* Read persistent properties after all default values have been loaded */
    load_persistent_properties();
    //创建 socket 服务（属性服务）
    fd = create_socket(PROP_SERVICE_NAME, SOCK_STREAM, 0666, 0, 0);
    if(fd < 0) return;
    fcntl(fd, F_SETFD, FD_CLOEXEC);
    fcntl(fd, F_SETFL, O_NONBLOCK);
    //开始服务监听
    listen(fd, 8);
    property_set_fd = fd;
}
```

通过上述代码可以知道属性服务的启动方式，另外在函数 start_property_service()中还涉及了如下两个宏。

☑ PROP_PATH_SYSTEM_BUILD
☑ PROP_PATH_SYSTEM_DEFAULT

上述两个宏都是系统预定义的属性文件名的路径，为了获取这些宏的定义，需要先分析函数 property_get()，该函数也是在 Property_service.c 中实现，具体实现代码如下所示。

```
const char* property_get(const char *name)
{
    prop_info *pi;
    if(strlen(name) >= PROP_NAME_MAX) return 0;
    pi = (prop_info*) __system_property_find(name);
    if(pi != 0) {
        return pi->value;
    } else {
        return 0;
    }
}
```

通过上述代码可以看到，在函数 property_get()中调用了核心函数__system_property_find()，该核心函数真正实现了获取属性值的功能。函数__system_property_find()属于 bionic 的一个 library，在文件 system_properties.c 中实现，可以在目录/bionic/libc/bionic 中找到该文件。

函数__system_property_find()的具体实现代码如下所示。

```
const prop_info *__system_property_find(const char *name)
{
    //获取属性存储内存区域的首地址
    prop_area *pa = __system_property_area__;
    unsigned count = pa->count;
    unsigned *toc = pa->toc;
    unsigned len = strlen(name);
    prop_info *pi;

    while(count--) {
        unsigned entry = *toc++;
        if(TOC_NAME_LEN(entry) != len) continue;

        pi = TOC_TO_INFO(pa, entry);
        if(memcmp(name, pi->name, len)) continue;
        return pi;
    }
    return 0;
}
```

从上述函数__system_property_find()的实现代码可以看出，第一行使用了一个__system_property_area__变量，该变量是全局的。

在文件 system_properties.c 对应的头文件 system_properties.h 中，定义了前面提到的两个表示属性文件路径的宏，其实还有另外两个表示路径的宏，共 4 个属性文件。文件 system_properties.h 可以在目



录/bionic/libc/include/sys 中找到。

这 4 个宏的具体定义如下所示。

```
#define PROP_PATH_RAMDISK_DEFAULT "/default.prop"
#define PROP_PATH_SYSTEM_BUILD "/system/build.prop"
#define PROP_PATH_SYSTEM_DEFAULT "/system/default.prop"
#define PROP_PATH_LOCAL_OVERRIDE "/data/local.prop"
```

此时可以进入 Android 设备的相应目录找到上述 4 个文件，一般会被保存在根目录中，通常在文件 default.prop 和 catdefault.prop 中会看到该文件的内容。而属性服务就是装载所有这 4 个属性文件中的所有属性以及使用 property_set 设置的属性。在 Android 设备的终端可以直接使用 getprop 命令从属性服务获取所有的属性值，如图 6-2 所示。另外，getprop 命令还可以直接根据属性获取具体的属性值，例如：

```
getprop ro.build.product
```

```
2|root@android:/dev/socket # getprop
[camera.flash_off]: [0]
[dalvik.vm.dexopt-flags]: [m=y]
[dalvik.vm.heapgrowthlimit]: [64m]
[dalvik.vm.heapmaxfree]: [8m]
[dalvik.vm.heapminfree]: [512k]
[dalvik.vm.heapsize]: [384m]
[dalvik.vm.heapstartsize]: [8m]
[dalvik.vm.heaptargetutilization]: [0.75]
[dalvik.vm.stack-trace-file]: [/data/anr/traces.txt]
[debug.nfc.fw_download]: [false]
[debug.nfc.se]: [false]
[dev.bootcomplete]: [1]
[dhcp.wlan0.dns1]: [192.168.17.254]
[dhcp.wlan0.dns2]: []
[dhcp.wlan0.dns3]: []
[dhcp.wlan0.dns4]: []
[dhcp.wlan0.gateway]: [192.168.17.254]
[dhcp.wlan0.ipaddress]: [192.168.17.104]
[dhcp.wlan0.leasetime]: [7200]
[dhcp.wlan0.mask]: [255.255.255.0]
[dhcp.wlan0.pid]: [10740]
[dhcp.wlan0.reason]: [RENEW]
[dhcp.wlan0.result]: [ok]
[dhcp.wlan0.server]: [192.168.17.254]
```

图 6-2 从属性服务获取所有的属性值

6.7.5 获取属性值

在 Android 5.0 源码中，getprop 命令的源代码文件是 getprop.c。读者可以在目录/system/core/toolbox/中找到该文件。

其实 getprop 获取属性值也是通过函数 property_get()完成的，此函数实际上调用了函数__system_property_find()，从__system_property_area__变量指定的内存区域获取相应的属性值。另外，在文件 system_properties.c 中还有如下两个函数用于通过属性服务修改或添加某个属性的值。

```
static int send_prop_msg(prop_msg *msg)
{
    struct pollfd pollfds[1];
    struct sockaddr_un addr;
    socklen_t alen;
    size_t namelen;
    int s;
    int r;
```

```
    int result = -1;
    //创建用于连接属性服务的 Socket
    s = socket(AF_LOCAL, SOCK_STREAM, 0);
    if(s < 0) {
        return result;
    }
    memset(&addr, 0, sizeof(addr));
    //property_service_socket 是 Socket 设备文件名称
    namelen = strlen(property_service_socket);
    strlcpy(addr.sun_path, property_service_socket, sizeof addr.sun_path);
    addr.sun_family = AF_LOCAL;
    alen = namelen + offsetof(struct sockaddr_un, sun_path) + 1;
    if(TEMP_FAILURE_RETRY(connect(s, (struct sockaddr *) &addr, alen)) < 0) {
        close(s);
        return result;
    }
    r = TEMP_FAILURE_RETRY(send(s, msg, sizeof(prop_msg), 0));
    if(r == sizeof(prop_msg)) {
        pollfds[0].fd = s;
        pollfds[0].events = 0;
        r = TEMP_FAILURE_RETRY(poll(pollfds, 1, 250 /* ms */));
        if (r == 1 && (pollfds[0].revents & POLLHUP) != 0) {
            result = 0;
        } else {


            result = 0;
        }
    }
    close(s);
    return result;
}
//用户可以直接调用该函数设置属性值
int __system_property_set(const char *key, const char *value)
{
    int err;
    int tries = 0;
    int update_seen = 0;
    prop_msg msg;
    if(key == 0) return -1;
    if(value == 0) value = "";
    if(strlen(key) >= PROP_NAME_MAX) return -1;
    if(strlen(value) >= PROP_VALUE_MAX) return -1;
    memset(&msg, 0, sizeof msg);
    msg.cmd = PROP_MSG_SETPROP;
    strlcpy(msg.name, key, sizeof msg.name);
    strlcpy(msg.value, value, sizeof msg.value);
    //设置属性值
    err = send_prop_msg(&msg);
```

```
    if(err < 0) {
        return err;
    }
    return 0;
}
```

在函数 send_prop_msg()中，涉及了重要变量 property_service_socket，具体定义如下所示。

```
static const char property_service_socket[] = "/dev/socket/" PROP_SERVICE_NAME;
```

实际上，send_prop_msg()是通过这个设备文件与属性服务通信的。读者可以在 Android 设备的终端进入/dev/socket 目录，通常会看到一个名为 property_service 的文件，该文件就是属性服务映射的设备文件。

6.7.6　处理请求

当属性服务器收到客户端的请求时，init 会调用函数 handle_property_set_fd()进行处理。当客户端的权限满足要求时，init 就调用函数 property_set()进行相关的处理。

```
int property_set(const char *name, const char *value)
{
    prop_area *pa;
    prop_info *pi;
    int namelen = strlen(name);
    int valuelen = strlen(value);
    if(namelen >= PROP_NAME_MAX) return -1;
    if(valuelen >= PROP_VALUE_MAX) return -1;
    if(namelen < 1) return -1;
    pi = (prop_info*) __system_property_find(name);
    if(pi != 0) {
        /* ro.* properties may NEVER be modified once set */
        if(!strncmp(name, "ro.", 3)) return -1;
        pa = __system_property_area__;
        update_prop_info(pi, value, valuelen);
        pa->serial++;
        __futex_wake(&pa->serial, INT32_MAX);
    } else {
        pa = __system_property_area__;
        if(pa->count == PA_COUNT_MAX) return -1;
        pi = pa_info_array + pa->count;
        pi->serial = (valuelen << 24);
        memcpy(pi->name, name, namelen + 1);
        memcpy(pi->value, value, valuelen + 1);
        pa->toc[pa->count] =
            (namelen << 24) | (((unsigned) pi) - ((unsigned) pa));
        pa->count++;
        pa->serial++;
        __futex_wake(&pa->serial, INT32_MAX);
    }
```

```
    /* If name starts with "net." treat as a DNS property. */
    if (strncmp("net.", name, strlen("net.")) == 0)   {
        if (strcmp("net.change", name) == 0) {
            return 0;
        }
        property_set("net.change", name);
    } else if (persistent_properties_loaded &&
            strncmp("persist.", name, strlen("persist.")) == 0) {
        write_persistent_property(name, value);
#ifdef HAVE_SELINUX
    } else if (strcmp("selinux.reload_policy", name) == 0 &&
                strcmp("1", value) == 0) {
        selinux_reload_policy();
#endif
    }
    property_changed(name, value);
    return 0;
}
```

到此为止，整个属性服务器的源码知识介绍完毕。

第 7 章　Zygote 进程详解

在 Android 5.0 系统中有如下 3 个十分重要的进程系统。

☑ Zygote 进程：被称为“孵化”进程或“孕育”进程，功能和 Linux 系统的 fork 类似，用于“孕育”产生出不同的子进程。
☑ System 进程：系统进程，是整个 Android Framework 所在的进程，用于启动 Android 系统。其核心进程是 system_server，其父进程就是 Zygote。
☑ 应用程序进程：每个 Android 应用程序运行后都会拥有自己的进程，这和 Windows 资源管理器中体现的进程是同一含义。

本章将详细分析 Android 5.0 中 Zygote 进程系统的基本知识。

7.1　Zygote 基础

Android 系统是基于 Linux 内核的，在 Linux 系统中的所有进程都是 init 进程的子孙进程。也就是说，所有进程都是直接或者间接地由 init 进程 fork（孕育）出来的。Zygote 进程也不例外，它是在系统启动的过程中由 init 进程创建的。Zygote 是 Android 系统的核心进程之一，被认为是 Android Framework 大家族的祖先。事实上，Zygote 正是平常所说的 Java 运行环境（JVM）。从总体架构上看，Zygote 是一个简单的典型 C/S 结构。其他进程作为一个客户端向 Zygote 发出“孕育”请求，当 Zygote 接收到命令后就“孕育”出一个 Activity 进程。具体“孕育”过程如图 7-1 所示。

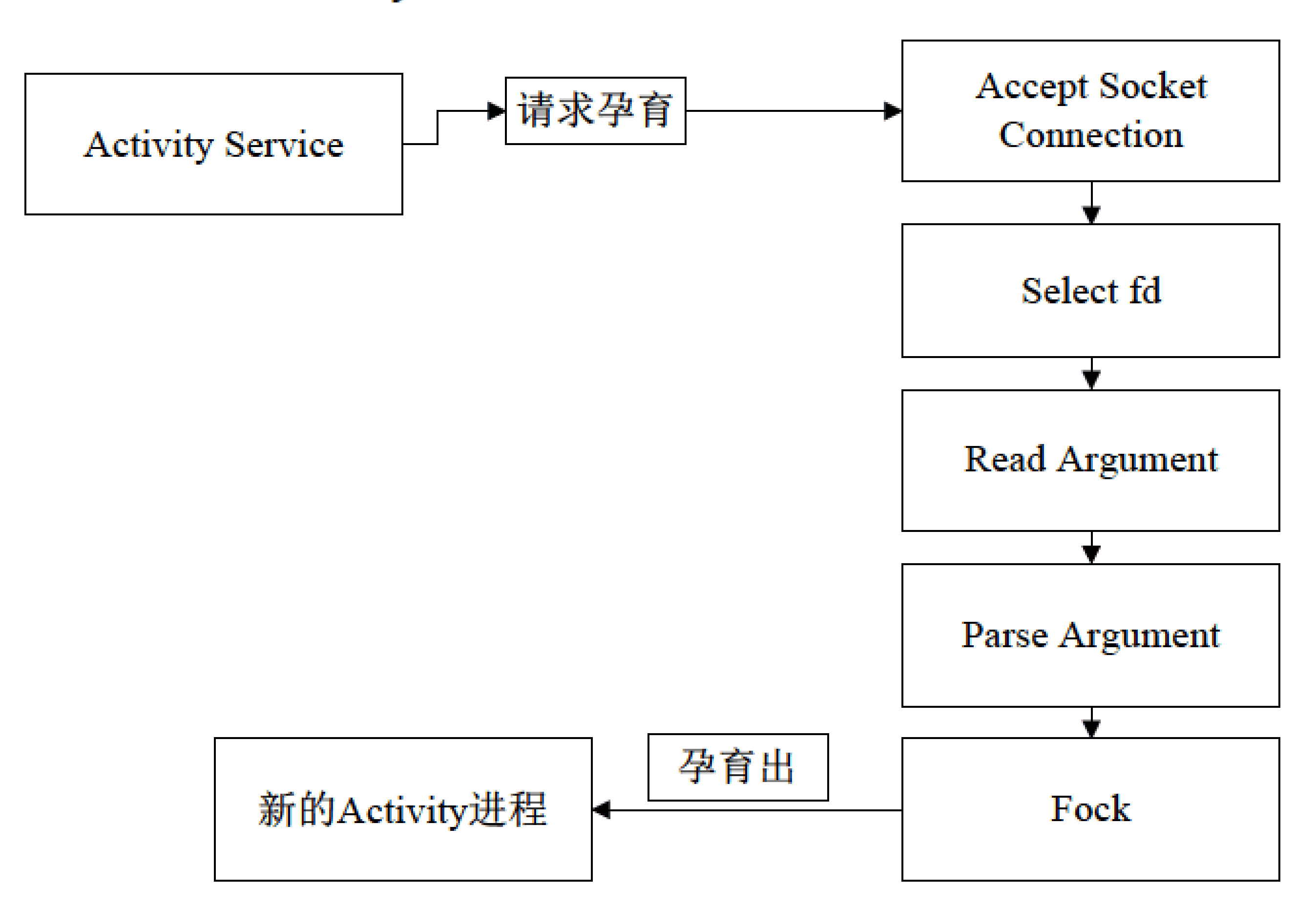


图 7-1　Zygote 的“孕育”过程

在 Android 系统中，如果查看进程列表，会发现进程 Zygote 的父进程是 init，而且它是所有应用的父进程；还有一个进程是 system_server，其父进程是 Zygote。其实 Zygote 服务实际上是一种 Select 服

务模型，是为启动 Java 代码而生，完成了一次 androidRuntime 的打开和关闭操作。Android 系统中的 Zygote 进程本身是一个应用层的程序，和驱动、内核模块等没有任何关系。Zygote 系统源码的组成及其调用结构如下所示。

（1）Zygote.java：提供访问 Dalvik 的 Zygote 接口，主要是包装 Linux 系统的 fork，以建立一个新的 VM 实例进程。

（2）ZygoteConnection.java：实现 Zygote 的套接口连接管理及其参数解析。其他 Actvitiy 建立进程请求是通过套接口发送命令参数给 Zygote。

（3）ZygoteInit.java：这是 Zygote 系统的 main()函数入口。

7.2 启动 Zygote

在 Android 5.0 源码中，在文件 system/core/rootdir/init.rc 中可以看到启动 Zygote 进程的脚本命令，具体代码如下所示。

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
    socket zygote stream 666
```

通过上述代码，系统启动后会在/dev/socket 目录下看到有一个名为 zygote 的文件。在上述代码中，相关关键字的具体说明如下所示。

- ☑ service：通知 init 进程创建一个名为 zygote 的进程，此 Zygote 进程要执行的程序是/system/bin/app_process，后面部分需要传给 app_process。
- ☑ socket：表示这个 Zygote 进程需要一个名称为 zygote 的 Socket 资源。

Zygote 最初的名字是 app_process，通过直接调用 pctrl 后把名字改成了 zygote。Zygote 进程执行的程序是 system/bin/app_process，其对应的源代码在文件 frameworks/base/cmds /app_process/app_main.cpp 中定义。

文件 app_main.cpp 的入口函数是 main()，本节将详细讲解启动 Zygote 进程的过程。

7.2.1 init.c 启动脚本

在文件 system/core/init/init.c 中，以服务的形式启动 Zygote 进程。在启动初始化进程 init 时，会调用函数 service_start()来启动 Zygote。函数 service_start()的具体实现代码如下所示。

```
void service_start(struct service *svc, const char *dynamic_args)
{
    struct stat s;
    pid_t pid;
    int needs_console;
    int n;
    char *scon = NULL;
    int rc;
    svc->flags &= (~(SVC_DISABLED|SVC_RESTARTING|SVC_RESET));
    svc->time_started = 0;
```

```
if (svc->flags & SVC_RUNNING) {
    return;
}

needs_console = (svc->flags & SVC_CONSOLE) ? 1 : 0;
if (needs_console && (!have_console)) {
    ERROR("service '%s' requires console\n", svc->name);
    svc->flags |= SVC_DISABLED;
    return;
}

if (stat(svc->args[0], &s) != 0) {
    ERROR("cannot find '%s', disabling '%s'\n", svc->args[0], svc->name);
    svc->flags |= SVC_DISABLED;
    return;
}

if ((!(svc->flags & SVC_ONESHOT)) && dynamic_args) {
    ERROR("service '%s' must be one-shot to use dynamic args, disabling\n",
           svc->args[0]);
    svc->flags |= SVC_DISABLED;
    return;
}

if (is_selinux_enabled() > 0) {
    if (svc->seclabel) {
        scon = strdup(svc->seclabel);
        if (!scon) {
            ERROR("Out of memory while starting '%s'\n", svc->name);
            return;
        }
    } else {
        char *mycon = NULL, *fcon = NULL;

        INFO("computing context for service '%s'\n", svc->args[0]);
        rc = getcon(&mycon);
        if (rc < 0) {
            ERROR("could not get context while starting '%s'\n", svc->name);
            return;
        }

        rc = getfilecon(svc->args[0], &fcon);
        if (rc < 0) {
            ERROR("could not get context while starting '%s'\n", svc->name);
            freecon(mycon);
            return;
        }

        rc = security_compute_create(mycon, fcon, string_to_security_class("process"), &scon);
        freecon(mycon);
        freecon(fcon);
```

```
            if (rc < 0) {
                ERROR("could not get context while starting '%s'\n", svc->name);
                return;
            }
        }
    }

    NOTICE("starting '%s'\n", svc->name);

    pid = fork();

    if (pid == 0) {
        struct socketinfo *si;
        struct svcenvinfo *ei;
        char tmp[32];
        int fd, sz;

        umask(077);
        if (properties_inited()) {
            get_property_workspace(&fd, &sz);
            sprintf(tmp, "%d,%d", dup(fd), sz);
            add_environment("ANDROID_PROPERTY_WORKSPACE", tmp);
        }

        for (ei = svc->envvars; ei; ei = ei->next)
            add_environment(ei->name, ei->value);

        setsockcreatecon(scon);

        for (si = svc->sockets; si; si = si->next) {
            int socket_type = (
                    !strcmp(si->type, "stream") ? SOCK_STREAM :
                        (!strcmp(si->type, "dgram") ? SOCK_DGRAM : SOCK_SEQPACKET));
            int s = create_socket(si->name, socket_type,
                                  si->perm, si->uid, si->gid);
            if (s >= 0) {
                publish_socket(si->name, s);
            }
        }

        freecon(scon);
        scon = NULL;
        setsockcreatecon(NULL);

        if (svc->ioprio_class != IoSchedClass_NONE) {
            if (android_set_ioprio(getpid(), svc->ioprio_class, svc->ioprio_pri)) {
                ERROR("Failed to set pid %d ioprio = %d,%d: %s\n",
                      getpid(), svc->ioprio_class, svc->ioprio_pri, strerror(errno));
            }
        }
```

```
        if (needs_console) {
            setsid();
            open_console();
        } else {
            zap_stdio();
        }

#if 0
        for (n = 0; svc->args[n]; n++) {
            INFO("args[%d] = '%s'\n", n, svc->args[n]);
        }
        for (n = 0; ENV[n]; n++) {
            INFO("env[%d] = '%s'\n", n, ENV[n]);
        }
#endif

        setpgid(0, getpid());

    /* as requested, set our gid, supplemental gids, and uid */
        if (svc->gid) {
            if (setgid(svc->gid) != 0) {
                ERROR("setgid failed: %s\n", strerror(errno));
                _exit(127);
            }
        }
        if (svc->nr_supp_gids) {
            if (setgroups(svc->nr_supp_gids, svc->supp_gids) != 0) {
                ERROR("setgroups failed: %s\n", strerror(errno));
                _exit(127);
            }
        }
        if (svc->uid) {
            if (setuid(svc->uid) != 0) {
                ERROR("setuid failed: %s\n", strerror(errno));
                _exit(127);
            }
        }
        if (svc->seclabel) {
            if (is_selinux_enabled() > 0 && setexeccon(svc->seclabel) < 0) {
                ERROR("cannot setexeccon('%s'): %s\n", svc->seclabel, strerror(errno));
                _exit(127);
            }
        }

        if (!dynamic_args) {
            if (execve(svc->args[0], (char**) svc->args, (char**) ENV) < 0) {
                ERROR("cannot execve('%s'): %s\n", svc->args[0], strerror(errno));
            }
        } else {
            char *arg_ptrs[INIT_PARSER_MAXARGS+1];
            int arg_idx = svc->nargs;
```

```
            char *tmp = strdup(dynamic_args);
            char *next = tmp;
            char *bword;

            /* Copy the static arguments */
            memcpy(arg_ptrs, svc->args, (svc->nargs * sizeof(char *)));

            while((bword = strsep(&next, " "))) {
                arg_ptrs[arg_idx++] = bword;
                if (arg_idx == INIT_PARSER_MAXARGS)
                    break;
            }
            arg_ptrs[arg_idx] = '\0';
            execve(svc->args[0], (char**) arg_ptrs, (char**) ENV);
        }
        _exit(127);
    }

    freecon(scon);

    if (pid < 0) {
        ERROR("failed to start '%s'\n", svc->name);
        svc->pid = 0;
        return;
    }

    svc->time_started = gettime();
    svc->pid = pid;
    svc->flags |= SVC_RUNNING;

    if (properties_inited())
        notify_service_state(svc->name, "running");
}
```

在上述代码中，每一个 Service 命令都会促使 init 进程调用 fork()函数来创建一个新的进程，在新的进程中会分析里面的 Socket 选项。对于每一个 Socket 选项来说，都会通过函数 create_socket()在/dev/socket 目录下创建一个文件。由此可见，函数 service_start()起到解释文件 init.rc 中 service 命令的作用。

7.2.2 创建一个 Socket

再看函数 create_socket()，其功能是调用函数 socket()创建一个 Socket，使用文件描述符 fd 来描述此 Socket。函数 create_socket()的具体实现代码如下所示。

```
int create_socket(const char *name, int type, mode_t perm, uid_t uid, gid_t gid)
{
    struct sockaddr_un addr;
    int fd, ret;
    char *secon;
    //调用函数 socket()创建一个 Socket，使用文件描述符 fd 来描述此 Socket
```

```
    fd = socket(PF_UNIX, type, 0);
    if (fd < 0) {
        ERROR("Failed to open socket '%s': %s\n", name, strerror(errno));
        return -1;
    }
    //为 Socket 创建一个类型为 AF_UNIX 的 Socket 地址 addr
    memset(&addr, 0 , sizeof(addr));
    addr.sun_family = AF_UNIX;
    snprintf(addr.sun_path, sizeof(addr.sun_path), ANDROID_SOCKET_DIR"/%s",
             name);
    ret = unlink(addr.sun_path);
    if (ret != 0 && errno != ENOENT) {
        ERROR("Failed to unlink old socket '%s': %s\n", name, strerror(errno));
        goto out_close;
    }
    secon = NULL;
    if (sehandle) {
        ret = selabel_lookup(sehandle, &secon, addr.sun_path, S_IFSOCK);
        if (ret == 0)
            setfscreatecon(secon);
    }
    ret = bind(fd, (struct sockaddr *) &addr, sizeof (addr));
    if (ret) {
        ERROR("Failed to bind socket '%s': %s\n", name, strerror(errno));
        goto out_unlink;
    }
    setfscreatecon(NULL);
    freecon(secon);
    //设置设备文件的/dev/socket/zygote 的用户 id、用户组 id 和用户权限
    chown(addr.sun_path, uid, gid);
    chmod(addr.sun_path, perm);
    INFO("Created socket '%s' with mode '%o', user '%d', group '%d'\n",
          addr.sun_path, perm, uid, gid);
    return fd;
out_unlink:
    unlink(addr.sun_path);
out_close:
    close(fd);
    return -1;
}
```

再看函数 publish_socket()，具体实现代码如下所示。

```
//参数 fd 是文件描述符，指向函数 create_socket()创建的 Socket
static void publish_socket(const char *name, int fd)
{
    char key[64] = ANDROID_SOCKET_ENV_PREFIX;
    char val[64];
    //将宏 ANDROID_SOCKET_ENV_PREFIX 和参数 name 描述的字符串连接在一起，并保存在字符串 key 中
    strlcpy(key + sizeof(ANDROID_SOCKET_ENV_PREFIX) - 1,
            name,
```

```
                sizeof(key) - sizeof(ANDROID_SOCKET_ENV_PREFIX));
    snprintf(val, sizeof(val), "%d", fd);
    add_environment(key, val);

    /* make sure we don't close-on-exec */
    fcntl(fd, F_SETFD, 0);
}
```

7.2.3 入口函数 main()

Zygote 的入口函数是 main()，功能是创建 AppRuntime 变量，然后调用成员函数 start()启动进程。函数 main()是在文件 frameworks/base/cmds/app_process/app_main.cpp 中定义的，具体实现代码如下所示。

```
int main(int argc, char* const argv[])
{
#ifdef __arm__
    char value[PROPERTY_VALUE_MAX];
    property_get("ro.kernel.qemu", value, "");
    bool is_qemu = (strcmp(value, "1") == 0);
    if ((getenv("NO_ADDR_COMPAT_LAYOUT_FIXUP") == NULL) && !is_qemu) {
        int current = personality(0xFFFFFFFF);
        if ((current & ADDR_COMPAT_LAYOUT) == 0) {
            personality(current | ADDR_COMPAT_LAYOUT);
            setenv("NO_ADDR_COMPAT_LAYOUT_FIXUP", "1", 1);
            execv("/system/bin/app_process", argv);
            return -1;
        }
    }
    unsetenv("NO_ADDR_COMPAT_LAYOUT_FIXUP");
#endif

    mArgC = argc;
    mArgV = argv;

    mArgLen = 0;
    for (int i=0; i<argc; i++) {
        mArgLen += strlen(argv[i]) + 1;
    }
    mArgLen--;

    AppRuntime runtime;
    const char* argv0 = argv[0];

    argc--;
    argv++;

    // Everything up to '--' or first non '-' arg goes to the vm
```

```
    int i = runtime.addVmArguments(argc, argv);

    bool zygote = false;
    bool startSystemServer = false;
    bool application = false;
    const char* parentDir = NULL;
    const char* niceName = NULL;
    const char* className = NULL;
    while (i < argc) {
        const char* arg = argv[i++];
        if (!parentDir) {
            parentDir = arg;
        } else if (strcmp(arg, "--zygote") == 0) {
            zygote = true;
            niceName = "zygote";
        } else if (strcmp(arg, "--start-system-server") == 0) {
            startSystemServer = true;
        } else if (strcmp(arg, "--application") == 0) {
            application = true;
        } else if (strncmp(arg, "--nice-name=", 12) == 0) {
            niceName = arg + 12;
        } else {
            className = arg;
            break;
        }
    }

    if (niceName && *niceName) {
        setArgv0(argv0, niceName);
        set_process_name(niceName);
    }

    runtime.mParentDir = parentDir;

    if (zygote) {
        runtime.start("com.android.internal.os.ZygoteInit",
                startSystemServer ? "start-system-server" : "");
    } else if (className) {
        runtime.mClassName = className;
        runtime.mArgC = argc - i;
        runtime.mArgV = argv + i;
        runtime.start("com.android.internal.os.RuntimeInit",
                application ? "application" : "tool");
    } else {
        fprintf(stderr, "Error: no class name or --zygote supplied.\n");
        app_usage();
        LOG_ALWAYS_FATAL("app_process: no class name or --zygote supplied.");
        return 10;
    }
}
```

7.2.4 启动函数创建一个虚拟机实例

Zygote 的启动函数是 start()，功能是调用函数 startVm()在 Zygote 中创建一个虚拟机实例。函数 start()是在文件 frameworks/base/core/jni/AndroidRuntime.cpp 中定义的，具体实现代码如下所示。

```
void AndroidRuntime::start(const char* className, const char* options)
{
    ALOGD("\n>>>>>> AndroidRuntime START %s <<<<<<\n",
            className != NULL ? className : "(unknown)");

    blockSigpipe();

    /*
     * 'startSystemServer == true' means runtime is obsolete and not run from
     * init.rc anymore, so we print out the boot start event here
     */
    if (strcmp(options, "start-system-server") == 0) {
        /* track our progress through the boot sequence */
        const int LOG_BOOT_PROGRESS_START = 3000;
        LOG_EVENT_LONG(LOG_BOOT_PROGRESS_START,
                       ns2ms(systemTime(SYSTEM_TIME_MONOTONIC)));
    }

    const char* rootDir = getenv("ANDROID_ROOT");
    if (rootDir == NULL) {
        rootDir = "/system";
        if (!hasDir("/system")) {
            LOG_FATAL("No root directory specified, and /android does not exist.");
            return;
        }
        setenv("ANDROID_ROOT", rootDir, 1);
    }

    //const char* kernelHack = getenv("LD_ASSUME_KERNEL");
    //ALOGD("Found LD_ASSUME_KERNEL='%s'\n", kernelHack);

    /*创建一个虚拟机实例*/
    JNIEnv* env;
    if (startVm(&mJavaVM, &env) != 0) {
        return;
    }
    onVmCreated(env);

    /*
     * 调用函数 startReg()在虚拟机实例中注册 JNI 方法
     */
    if (startReg(env) < 0) {
        ALOGE("Unable to register all android natives\n");
        return;
```

```
    }
    jclass stringClass;
    jobjectArray strArray;
    jstring classNameStr;
    jstring optionsStr;

    stringClass = env->FindClass("java/lang/String");
    assert(stringClass != NULL);
    strArray = env->NewObjectArray(2, stringClass, NULL);
    assert(strArray != NULL);
    classNameStr = env->NewStringUTF(className);
    assert(classNameStr != NULL);
    env->SetObjectArrayElement(strArray, 0, classNameStr);
    optionsStr = env->NewStringUTF(options);
    env->SetObjectArrayElement(strArray, 1, optionsStr);
    char* slashClassName = toSlashClassName(className);
    jclass startClass = env->FindClass(slashClassName);
    if (startClass == NULL) {
        ALOGE("JavaVM unable to locate class '%s'\n", slashClassName);
        /* keep going */
    } else {
        jmethodID startMeth = env->GetStaticMethodID(startClass, "main",
            "([Ljava/lang/String;)V");
        if (startMeth == NULL) {
            ALOGE("JavaVM unable to find main() in '%s'\n", className);
            /* keep going */
        } else {
//调用类 com.android.internal.os.ZygoteInit 的静态成员函数 main()来启动 Zygote 进程
            env->CallStaticVoidMethod(startClass, startMeth, strArray);

#if 0
            if (env->ExceptionCheck())
                threadExitUncaughtException(env);
#endif
        }
    }
    free(slashClassName);
    ALOGD("Shutting down VM\n");
    if (mJavaVM->DetachCurrentThread() != JNI_OK)
        ALOGW("Warning: unable to detach main thread\n");
    if (mJavaVM->DestroyJavaVM() != 0)
        ALOGW("Warning: VM did not shut down cleanly\n");
}
```

在上述代码中，通过调用类 com.android.internal.os.ZygoteInit 中的函数 main()启动了 Zygote 进程。此成员函数在文件 frameworks/base/core/java/com/android/internal/os/ZygoteInit.java 中定义，具体实现代码如下所示。

```
public static void main(String argv[]) {
    try {
```

```
            SamplingProfilerIntegration.start();
            //调用函数 registerZygoteSocket()创建一个 Socket 接口
            registerZygoteSocket();
            EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_START,
                SystemClock.uptimeMillis());
            preload();
            EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_END,
                SystemClock.uptimeMillis());

            SamplingProfilerIntegration.writeZygoteSnapshot();

            gc();

            Trace.setTracingEnabled(false);

            if (argv.length != 2) {
                throw new RuntimeException(argv[0] + USAGE_STRING);
            }
            //调用函数 startSystemServer()启动 SystemServer 组件
            if (argv[1].equals("start-system-server")) {
                startSystemServer();
            } else if (!argv[1].equals("")) {
                throw new RuntimeException(argv[0] + USAGE_STRING);
            }

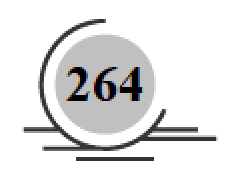
            Log.i(TAG, "Accepting command socket connections");
            //调用函数 runSelectLoop()进入一个无限循环
            //在前面创建的 Socket 接口中等待 ActivityManagerService 请求，以创建新的应用程序进程
            runSelectLoop();

            closeServerSocket();
        } catch (MethodAndArgsCaller caller) {
            caller.run();
        } catch (RuntimeException ex) {
            Log.e(TAG, "Zygote died with exception", ex);
            closeServerSocket();
            throw ex;
        }
}
```

7.2.5 和 Zygote 进程中的 Socket 实现连接

在 Android 系统中，ActivityManagerService 通过函数 Process.start()创建一个新的进程。函数 Process.start()会先通过 Socket 连接到 Zygote 进程，并由 Zygote 进程实现创建新应用程序进程的功能。另外，类 Process 是通过函数 openZygoteSocketIfNeeded()连接到 Zygote 进程中的 Socket 的。函数 openZygoteSocketIfNeeded()在文件 frameworks/base/core/java/android/os/Process.java 中定义，具体实现代码如下所示。

```
private static void openZygoteSocketIfNeeded()
            throws ZygoteStartFailedEx {
```



```
    int retryCount;
    if (sPreviousZygoteOpenFailed) {
        retryCount = 0;
    } else {
        retryCount = 10;
    }
    for (int retry = 0
            ; (sZygoteSocket == null) && (retry < (retryCount + 1))
            ; retry++ ) {
        if (retry > 0) {
            try {
                Log.i("Zygote", "Zygote not up yet, sleeping...");
                Thread.sleep(ZYGOTE_RETRY_MILLIS);
            } catch (InterruptedException ex) {
            }
        }
        try {
            sZygoteSocket = new LocalSocket();
            sZygoteSocket.connect(new LocalSocketAddress(ZYGOTE_SOCKET,
                    LocalSocketAddress.Namespace.RESERVED));
            sZygoteInputStream
                    = new DataInputStream(sZygoteSocket.getInputStream());
            sZygoteWriter =
                new BufferedWriter(
                        new OutputStreamWriter(
                                sZygoteSocket.getOutputStream()),
                        256);
            Log.i("Zygote", "Process: zygote socket opened");
            sPreviousZygoteOpenFailed = false;
            break;
        } catch (IOException ex) {
            if (sZygoteSocket != null) {
                try {
                    sZygoteSocket.close();
                } catch (IOException ex2) {
                    Log.e(LOG_TAG,"I/O exception on close after exception",
                            ex2);
                }
            }
            sZygoteSocket = null;
        }
    }
    if (sZygoteSocket == null) {
        sPreviousZygoteOpenFailed = true;
        throw new ZygoteStartFailedEx("connect failed");
    }
}
```

在文件 ZygoteInit.java 中的函数 main()的实现代码中，用到了函数 registerZygoteSocket()，此函数在文件 frameworks/base/core/java/com/android/internal/os/ZygoteInit.java 中定义，具体实现代码如下所示。

```
private static void registerZygoteSocket() {
    if (sServerSocket == null) {
        int fileDesc;
        try {
            String env = System.getenv(ANDROID_SOCKET_ENV);
            fileDesc = Integer.parseInt(env);
        } catch (RuntimeException ex) {
            throw new RuntimeException(
                ANDROID_SOCKET_ENV + " unset or invalid", ex);
        }

        try {
            sServerSocket = new LocalServerSocket(
                createFileDescriptor(fileDesc));
        } catch (IOException ex) {
            throw new RuntimeException(
                "Error binding to local socket '" + fileDesc + "'", ex);
        }
    }
}
```

在上述代码中，通过文件描述符创建了 Socket 接口，此文件描述符就是本书前面所介绍的文件/dev/socket/zygote，而此文件描述符通过环境变量 ANDROID_SOCKET_ENV 获得。另外，由 init 进程负责解释执行系统启动脚本文件 system/core/rootdir/init.rc，而 init 进程的源代码位于文件 system/core/init/init.c 中，由函数 service_start()负责解释文件 init.rc 中的 service 命令。在 service_start()函数中，每一个 service 命令都会促使进程 init 调用函数 fork()创建一个新的进程，在新进程中会解析里面的 socket 选项。对于每一个 socket 选项来说，全部会通过函数 create_socke()在/dev/socket 目录下创建一个 Zygote 文件，然后通过函数 publish_socket()将得到的文件描述符写入到环境变量中。函数 publish_socket()的具体实现代码如下所示。

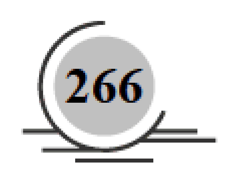
```
static void publish_socket(const char *name, int fd)
{
    char key[64] = ANDROID_SOCKET_ENV_PREFIX;
    char val[64];

    strlcpy(key + sizeof(ANDROID_SOCKET_ENV_PREFIX) - 1,
            name,
            sizeof(key) - sizeof(ANDROID_SOCKET_ENV_PREFIX));
    snprintf(val, sizeof(val), "%d", fd);
    add_environment(key, val);

    /* make sure we don't close-on-exec */
    fcntl(fd, F_SETFD, 0);
}
```

在上述代码中，传进的参数 name 值为 zygote，而 ANDROID_SOCKET_ENV_PREFIX 在文件 system/core/include/cutils/sockets.h 中的定义代码如下。

```
#define ANDROID_SOCKET_ENV_PREFIX "ANDROID_SOCKET_"
```



这样就将得到的文件描述符写入到 ANDROID_SOCKET_zygote 环境变量，这个环境变量值为 key 值。因为函数 ZygoteInit.registerZygoteSocket()和 create_socket()都运行在同一个进程中，所以函数 ZygoteInit.registerZygoteSocket()可以直接使用文件描述符来创建一个 Java 层的 LocalServerSocket 对象。如果其他进程也需要打开/dev/socket/zygote 文件和 Zygote 进程进行通信，则必须通过文件名作为中介来连接 LocalServerSocket。

在文件 ZygoteInit.java 中的函数 main()的实现代码中，用到了函数 startSystemServer()，此函数也是在文件 frameworks/base/core/java/com/android/internal/os/ZygoteInit.java 中定义的，具体实现代码如下所示。

```
private static boolean startSystemServer()
        throws MethodAndArgsCaller, RuntimeException {
    /* Hardcoded command line to start the system server */
    String args[] = {
        "--setuid=1000",
        "--setgid=1000",
"--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1018,3001,3002,3003,3006,3007",
        "--capabilities=130104352,130104352",
        "--runtime-init",
        "--nice-name=system_server",
        "com.android.server.SystemServer",
    };
    ZygoteConnection.Arguments parsedArgs = null;

    int pid;

    try {
        parsedArgs = new ZygoteConnection.Arguments(args);
        ZygoteConnection.applyDebuggerSystemProperty(parsedArgs);
        ZygoteConnection.applyInvokeWithSystemProperty(parsedArgs);

         /* Request to fork the system server process */
        pid = Zygote.forkSystemServer(
                parsedArgs.uid, parsedArgs.gid,
                parsedArgs.gids,
                parsedArgs.debugFlags,
                null,
                parsedArgs.permittedCapabilities,
                parsedArgs.effectiveCapabilities);
    } catch (IllegalArgumentException ex) {
        throw new RuntimeException(ex);
    }
    /* For child process */
    if (pid == 0) {
        handleSystemServerProcess(parsedArgs);
    }
    return true;
}
```

在文件 ZygoteInit.java 中的函数 main()的实现代码中，用到了函数 startSystemServer()，此函数在文件 frameworks/base/core/java/com/android/internal/os/ZygoteInit.java 中定义，具体实现代码如下所示。

```
private static boolean startSystemServer()
        throws MethodAndArgsCaller, RuntimeException {
    /* Hardcoded command line to start the system server */
    String args[] = {
        "--setuid=1000",
        "--setgid=1000",
        "--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1018,3001,3002,3003,3006,3007",
        "--capabilities=130104352,130104352",
        "--runtime-init",
        "--nice-name=system_server",
        "com.android.server.SystemServer",
    };
    ZygoteConnection.Arguments parsedArgs = null;

    int pid;

    try {
        parsedArgs = new ZygoteConnection.Arguments(args);
        ZygoteConnection.applyDebuggerSystemProperty(parsedArgs);
        ZygoteConnection.applyInvokeWithSystemProperty(parsedArgs);
        /* Request to fork the system server process */
        pid = Zygote.forkSystemServer(
                parsedArgs.uid, parsedArgs.gid,
                parsedArgs.gids,
                parsedArgs.debugFlags,
                null,
                parsedArgs.permittedCapabilities,
                parsedArgs.effectiveCapabilities);
    } catch (IllegalArgumentException ex) {
        throw new RuntimeException(ex);
    }
    /* For child process */
    if (pid == 0) {
        handleSystemServerProcess(parsedArgs);
    }
    return true;
}
```

在上述代码中，Zygote 进程通过函数 forkSystemServer()“孕育”了一个新的进程来启动 SystemServer 组件，返回值 pid 为 0 的位置标示新进程的执行路径，即新建进程会执行函数 handleSystemServerProcess()。函数 handleSystemServerProcess()在文件 frameworks/base/core/java/com/android/internal/os/ZygoteInit.java 中定义，具体实现代码如下所示。

```
private static void handleSystemServerProcess(
        ZygoteConnection.Arguments parsedArgs)
        throws ZygoteInit.MethodAndArgsCaller {
    closeServerSocket();
    Libcore.os.umask(S_IRWXG | S_IRWXO);
    if (parsedArgs.niceName != null) {
        Process.setArgV0(parsedArgs.niceName);
```

```
    }
    if (parsedArgs.invokeWith != null) {
        WrapperInit.execApplication(parsedArgs.invokeWith,
                parsedArgs.niceName, parsedArgs.targetSdkVersion,
                null, parsedArgs.remainingArgs);
    } else {
        /*
         * Pass the remaining arguments to SystemServer
         */
        RuntimeInit.zygoteInit(parsedArgs.targetSdkVersion, parsedArgs.remainingArgs);
    }
    /* should never reach here */
}
```

在上述代码中，调用函数 closeServerSocket()关闭了子进程，然后调用函数 RuntimeInit.zygoteInit()进一步启动 SystemServer 组件。函数 RuntimeInit.zygoteInit()在文件 frameworks/base/core/java/com/android/internal/os/RuntimeInit.java 中定义，具体实现代码如下所示。

```
public static final void zygoteInit(int targetSdkVersion, String[] argv)
        throws ZygoteInit.MethodAndArgsCaller {
    if (DEBUG) Slog.d(TAG, "RuntimeInit: Starting application from zygote");

    redirectLogStreams();
    commonInit();
//调用函数 zygoteInitNative()来执行一个 Binder 进程间通信机制的初始化工作
//当完成这个工作后，这个进程中的 Binder 对象就可以方便地进行进程间通信
    nativeZygoteInit();

    applicationInit(targetSdkVersion, argv);
}
```

在文件 ZygoteInit.java 中的函数 main()的实现代码中，用到了函数 runSelectLoop()，功能是进入一个无限循环在前面创建的 Socket 接口中，并等待 ActivityManagerService 请求创建新的应用程序进程。函数 runSelectLoop()在文件 frameworks/base/core/java/com/android/internal/os/ZygoteInit.java 中定义，具体实现代码如下所示。

```
private static void runSelectLoop() throws MethodAndArgsCaller {
    ArrayList<FileDescriptor> fds = new ArrayList<FileDescriptor>();
    ArrayList<ZygoteConnection> peers = new ArrayList<ZygoteConnection>();
    FileDescriptor[] fdArray = new FileDescriptor[4];

    fds.add(sServerSocket.getFileDescriptor());
    peers.add(null);

    int loopCount = GC_LOOP_COUNT;
    while (true) {
        int index;

        /*
         * Call gc() before we block in select().
         * It's work that has to be done anyway, and it's better
```

```
              * to avoid making every child do it.   It will also
              * madvise() any free memory as a side-effect.
              *
              * Don't call it every time, because walking the entire
              * heap is a lot of overhead to free a few hundred bytes
              */
            if (loopCount <= 0) {
                gc();
                loopCount = GC_LOOP_COUNT;
            } else {
                loopCount--;
            }


            try {
                fdArray = fds.toArray(fdArray);
                index = selectReadable(fdArray);
            } catch (IOException ex) {
                throw new RuntimeException("Error in select()", ex);
            }

            if (index < 0) {
                throw new RuntimeException("Error in select()");
            } else if (index == 0) {
                ZygoteConnection newPeer = acceptCommandPeer();
                peers.add(newPeer);
                fds.add(newPeer.getFileDesciptor());
            } else {
                boolean done;
//将数据通过 Socket 接口发送出去后会执行下面的语句
//peers.get(index)得到的是一个 ZygoteConnection 对象，表示一个 Socket 连接
//调用 ZygoteConnection.runOnce()函数进一步处理
done = peers.get(index).runOnce();

                if (done) {
                    peers.remove(index);
                    fds.remove(index);
                }
            }
        }
}
```

通过上述代码，可以等待 ActivityManagerService 连接这个 Socket，然后调用函数 ZygoteConnection.runOnce()创建新的应用程序。

第 8 章　System 进程详解

在 Android 5.0 系统中，System 进程和系统服务有着重要的关系。几乎所有的 Android 核心服务都在这个进程中，例如，ActivityManagerService、PowerManagerService 和 WindowManagerService 等。System 进程是系统进程，也是整个 Android Framework 所在的进程，用于启动 Android 系统。其核心进程是 system_server，其父进程就是 Zygote。本章将详细分析 Android 5.0 中的 System 进程的核心知识和具体架构原理。

8.1　启动前的准备

在 Android 5.0 系统中，通过静态类 ZygoteInit 的成员函数 handleSystemServerProcess()来启动 System 进程。具体启动过程如图 8-1 所示。



图 8-1　启动 System 进程前的准备工作

本节将详细讲解在启动 System 进程之前需要做的准备工作。

8.1.1　获取创建的 Socket

首先在文件 frameworks/base/core/java/com/android/internal/os/ZygoteInit.java 中，获取 Zygote 进程在启动过程中创建的 Socket。其实 System 进程不需要这个 Socket，所以会调用类 ZygoteInit 的成员函数 closeServerSocket()关闭这个 Socket。对应的代码如下所示。

```
private static void handleSystemServerProcess(
            ZygoteConnection.Arguments parsedArgs)
            throws ZygoteInit.MethodAndArgsCaller {
    //关闭这个 Socket
    closeServerSocket();
    Libcore.os.umask(S_IRWXG | S_IRWXO);
    if (parsedArgs.niceName != null) {
        Process.setArgV0(parsedArgs.niceName);
    }
    if (parsedArgs.invokeWith != null) {
        WrapperInit.execApplication(parsedArgs.invokeWith,
```

```
                        parsedArgs.niceName, parsedArgs.targetSdkVersion,
                        null, parsedArgs.remainingArgs);
    } else {
        /*
         * Pass the remaining arguments to SystemServer.
         */
        //调用类 RuntimeInit 的静态函数 zygoteInit()启动 System 进程
        RuntimeInit.zygoteInit(parsedArgs.targetSdkVersion, parsedArgs.remainingArgs);
    }
    /* should never reach here */
}
```

8.1.2 启动 System 进程

接下来调用类 RuntimeInit 的静态函数 zygoteInit()启动 System 进程，此函数在文件 frameworks/base/core/java/com/android/internal/os/ZygoteInit.java 中定义，具体实现代码如下所示。

```
public static final void zygoteInit(int targetSdkVersion, String[] argv)
        throws ZygoteInit.MethodAndArgsCaller {
    if (DEBUG) Slog.d(TAG, "RuntimeInit: Starting application from zygote");

    redirectLogStreams();
    //调用函数 commonInit()设置 System 进程的时区和键盘布局等信息
    commonInit ();
    //调用函数 nativeZygoteInit()启动一个 Binder 线程池
    nativeZygoteInit();

    applicationInit(targetSdkVersion, argv);
}
```

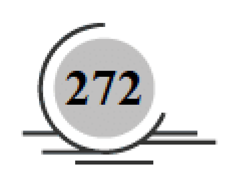
8.2 分析 SystemServer

SystemServer 是 Android 中 Java 层的两大支柱进程之一，另一个是专门负责孵化 Java 进程的 Zygote。如果这两大支柱其中的任何一个崩溃了，都会导致 Java 层的崩溃。如果 Java 层真的崩溃了，则 Linux 系统中的进程 init 会重新启动 SystemServer 和 Zygote，以重新建立 Android 的 Java 层。本节将首先详细分析 SystemServer 的核心源码。

8.2.1 分析主函数 main()

☑ SystemServer 是由 Zygote 孵化而来的一个进程，通过 ps 命令，可知其进程名为 system_server。在 DDMS 中可以看到，进程 system_server 的进程名为 system_process。SystemServer 核心逻辑的入口是函数 main()，此入口函数在文件/frameworks/base/services/java/com/android/server/SystemServer.java 中实现。

文件 SystemServer.java 的入口函数是 main()，具体实现代码如下所示。



```
public static void main(String[] args) {
        if (System.currentTimeMillis() < EARLIEST_SUPPORTED_TIME) {
            //如果系统时钟早于 1970 年，则设置系统时钟从 1970 年开始
            Slog.w(TAG, "System clock is before 1970; setting to 1970.");
            SystemClock.setCurrentTimeMillis(EARLIEST_SUPPORTED_TIME);
        }

        if (SamplingProfilerIntegration.isEnabled()) {
            SamplingProfilerIntegration.start();
            timer = new Timer();
            timer.schedule(new TimerTask() {
                @Override
                public void run() {
                    //SystemServer 性能统计，每小时统计一次，统计结果输出为文件
                    SamplingProfilerIntegration.writeSnapshot("system_server", null);
                } // SNAPSHOT_INTERVAL 定义为 1 小时
            }, SNAPSHOT_INTERVAL, SNAPSHOT_INTERVAL);
        }

        //和 Dalvik 虚拟机相关的设置，主要是内存使用方面的控制
        dalvik.system.VMRuntime.getRuntime().clearGrowthLimit();

        // The system server has to run all of the time, so it needs to be
        // as efficient as possible with its memory usage
        VMRuntime.getRuntime().setTargetHeapUtilization(0.8f);
        //加载动态库 libandroid_servers.so
        System.loadLibrary("android_servers");
        init1(args); //调用 native 的 init1()函数
    }

    public static final void init2() {
        Slog.i(TAG, "Entered the Android system server!");
        Thread thr = new ServerThread();
        thr.setName("android.server.ServerThread");
        thr.start();
    }
}
```

由此可见，函数 main()首先做一些初始化工作，然后加载动态库 libandroid_servers.so，最后调用 native 的函数 init1()。该函数在 libandroid_servers.so 库中实现，在文件/frameworks/base/services/jni/com_android_server_SystemServer.cpp 中定义。

函数 init1()的具体实现代码如下所示。

```
extern "C" int system_init();
static void android_server_SystemServer_init1(JNIEnv* env, jobject clazz)
{
    system_init();              //调用上面用 extern  声明的 system_init()函数
}
```

而函数 system_init()在另外一个库 libsystem_server.so 中实现，在文件\frameworks\base\cmds\system_

server\library\System_init.cpp 中定义。

函数 system_init()的具体实现代码如下所示。

```
extern "C" status_t system_init()
{
    ALOGI("Entered system_init()");

    sp<ProcessState> proc(ProcessState::self());

    sp<IServiceManager> sm = defaultServiceManager();
    ALOGI("ServiceManager: %p\n", sm.get());

    sp<GrimReaper> grim = new GrimReaper();
    sm->asBinder()->linkToDeath(grim, grim.get(), 0);

    char propBuf[PROPERTY_VALUE_MAX];
    property_get("system_init.startsurfaceflinger", propBuf, "1");
    if (strcmp(propBuf, "1") == 0) {
        SurfaceFlinger::instantiate();
    }

    property_get("system_init.startsensorservice", propBuf, "1");
    if (strcmp(propBuf, "1") == 0) {
        SensorService::instantiate();
    }

    ALOGI("System server: starting Android runtime.\n");
    AndroidRuntime* runtime = AndroidRuntime::getRuntime();

    ALOGI("System server: starting Android services.\n");
    JNIEnv* env = runtime->getJNIEnv();
    if (env == NULL) {
        return UNKNOWN_ERROR;
    }
    jclass clazz = env->FindClass("com/android/server/SystemServer");
    if (clazz == NULL) {
        return UNKNOWN_ERROR;
    }
    jmethodID methodId = env->GetStaticMethodID(clazz, "init2", "()V");
    if (methodId == NULL) {
        return UNKNOWN_ERROR;
    }
    env->CallStaticVoidMethod(clazz, methodId);

    ALOGI("System server: entering thread pool.\n");
    ProcessState::self()->startThreadPool();
    IPCThreadState::self()->joinThreadPool();
    ALOGI("System server: exiting thread pool.\n");

    return NO_ERROR;
}
```

通过上述代码可知，SystemServer 中的函数 main()通过函数 init1()，从 Java 层穿越到 Native 层，实现了一些初始化工作后，又通过 JNI 从 Native 层穿越到 Java 层去调用函数 init2()。函数 init2()返回后，最终又回归到 Native 层。

8.2.2　分析函数 init2()

在文件 SystemServer.java 中，函数 init1()较简单，其实重点内容都在函数 init2()中。函数 init2()的具体实现代码如下所示。

```
public static final void init2() {
        Thread thr = new ServerThread();
        thr.setName("android.server.ServerThread");
        thr.start();//启动一个线程，此线程中包含了众多 Service
}
```

通过上述代码将创建一个新的线程 ServerThread，该线程的 run()函数的实现代码有 600 多行，如此之长的原因是 Android 平台中众多 Service 都汇集于此。

在 Android 平台中，共有 7 大类 43 个 Service（包括 Watchdog）。实际上，还有一些 Service 并没有在 ServerThread 的 run()函数中出现。这 7 大类服务如下所示。

- ☑ 第 1 大类：是 Android 的核心服务，如 ActivityManagerService、WindowManager-Service 等。
- ☑ 第 2 大类：是和通信相关的服务，如 Wi-Fi 相关服务、Telephone 相关服务。
- ☑ 第 3 大类：是和系统功能相关的服务，如 AudioService、MountService 和 Usb-Service 等。
- ☑ 第 4 大类：是 BatteryService、VibratorService 等服务。
- ☑ 第 5 大类：是 EntropyService、DiskStatsService 和 Watchdog 等相对独立的服务。
- ☑ 第 6 大类：是蓝牙服务。
- ☑ 第 7 大类：是和 UI 紧密相关的服务，如状态栏服务、通知管理服务等。

在本章后面的内容中，将详细分析其中的第 5 类服务。该类中的 Service 之间关系简单，而且功能相对独立。第 5 大类服务包括如下服务。

- ☑ EntropyService：熵服务，和随机数的生成有关。
- ☑ ClipboardService：剪贴板服务。
- ☑ DropBoxManagerService：该服务和系统运行时日志的存储与管理有关。
- ☑ DiskStatsService 和 DeviceStorageMonitorService：这两个服务用于查看和监测系统存储空间。
- ☑ SamplingProfilerService：这个服务是从 Android 4.0 新增的，功能非常简单。
- ☑ Watchdog：即看门狗，是 Android 的“老员工”了。Android 2.3 以后其内存检测功能被去掉，所以与 Android 2.2 相比，显得更简单了。

8.3　第一个启动的 ServiceEntropyService

EntropyService 是 SystemServer 启动的第一个 Service，它以 3 个小时为单位周期性加载和保存熵池（/dev/urandom）。但是由于/dev/urandom 本身就有的安全性要比/dev/random 相对差些，所以每隔 3

小时，Android 系统在 kernel 的熵池中增加一些附加信息，这些信息对提高随机数的质量是有帮助的。Android 会添加如下额外信息。

- ☑ out.println("Copyright (C) 2009 The Android Open Source Project");
- ☑ out.println("All Your Randomness Are Belong To Us");
- ☑ out.println(START_TIME);
- ☑ out.println(START_NANOTIME);
- ☑ out.println(SystemProperties.get("ro.serialno"));
- ☑ out.println(SystemProperties.get("ro.bootmode"));
- ☑ out.println(SystemProperties.get("ro.baseband"));
- ☑ out.println(SystemProperties.get("ro.carrier"));
- ☑ out.println(SystemProperties.get("ro.bootloader"));
- ☑ out.println(SystemProperties.get("ro.hardware"));
- ☑ out.println(SystemProperties.get("ro.revision"));
- ☑ out.println(System.currentTimeMillis());
- ☑ out.println(System.nanoTime());

根据物理学基本原理，一个系统的熵越大，该系统就越不稳定。在 Android 中，目前也只有随机数常处于这种不稳定的系统中。在 Android 系统中，SystemServer 中添加该服务的代码如下所示。

```
ServiceManager.addService("entropy", new EntropyService());
```

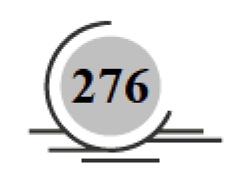
上述代码非常简单，从中可直接分析 EntropyService 的构造函数，此函数在文件 EntropyService.java 中定义，具体实现代码如下所示。

```
public EntropyService() {
        //调用另外一个构造函数，getSystemDir()函数返回的是/data/system 目录
        this(getSystemDir() + "/entropy.dat", "/dev/urandom");
}
public EntropyService(String entropyFile, String randomDevice) {
  this.randomDevice = randomDevice;//urandom 是 Linux 系统中产生随机数的设备
  // /data/system/entropy.dat 文件保存了系统此前的熵信息
  this.entropyFile = entropyFile;
  //下面有 4 个关键函数
  loadInitialEntropy();
  addDeviceSpecificEntropy();
  writeEntropy();
  scheduleEntropyWriter();
}
```

从以上代码中可以看出，EntropyService 构造函数中依次调用了 4 个关键函数，这 4 个函数比较简单。本节将详细讲解这 4 个函数的基本知识。

8.3.1 将内容写到 urandom 设备

函数 loadInitialEntropy()的功能是将文件 entropy.dat 的内容写到 urandom 设备，这样可增加系统的随机性。在系统中有一个 entropy pool，在刚启动系统时，该 pool 中的内容为空，会导致早期生成的随



机数变得可预测。通过将 entropy.dat 数据写到该 entropy pool（这样该 pool 中的内容就不为空）中，随机数的生成就无规律可言了。函数 loadInitialEntropy()的具体实现代码如下所示。

```
private void loadInitialEntropy() {
    try {
        RandomBlock.fromFile(entropyFile).toFile(randomDevice);
    } catch (IOException e) {
        Slog.w(TAG, "unable to load initial entropy (first boot?)", e);
    }
}
```

8.3.2　将和设备相关的信息写到 urandom 设备

函数 addDeviceSpecificEntropy()的功能是将一些和设备相关的信息写入 urandom 设备，具体实现代码如下所示。

```
private void addDeviceSpecificEntropy() {
    PrintWriter out = null;
    try {
        out = new PrintWriter(new FileOutputStream(randomDevice));
        out.println("Copyright (C) 2009 The Android Open Source Project");
        out.println("All Your Randomness Are Belong To Us");
        out.println(START_TIME);
        out.println(START_NANOTIME);
        out.println(SystemProperties.get("ro.serialno"));
        out.println(SystemProperties.get("ro.bootmode"));
        out.println(SystemProperties.get("ro.baseband"));
        out.println(SystemProperties.get("ro.carrier"));
        out.println(SystemProperties.get("ro.bootloader"));
        out.println(SystemProperties.get("ro.hardware"));
        out.println(SystemProperties.get("ro.revision"));
        out.println(System.currentTimeMillis());
        out.println(System.nanoTime());
    } catch (IOException e) {
        Slog.w(TAG, "Unable to add device specific data to the entropy pool", e);
    } finally {
        if (out != null) {
            out.close();
        }
    }
}
```

由上述代码可知，即使向 urandom 的 entropy pool 中写入了固定信息，也能增加随机数生成的随机性。从熵的角度考虑，系统的质量越大（即 pool 中的内容越多），该系统就越不稳定。

8.3.3　读取 urandom 设备的内容

函数 writeEntropy()的功能是读取 urandom 设备的内容到 entropy.dat 文件。具体实现代码如下所示。

```
private void writeEntropy() {
    try {
        RandomBlock.fromFile(randomDevice).toFile(entropyFile);
    } catch (IOException e) {
        Slog.w(TAG, "unable to write entropy", e);
    }
}
```

8.3.4 发送 ENTROPY_WHAT

函数 scheduleEntropyWriter()的功能是向 EntropyService 内部的 Handler 发送一个 ENTROPY_WHAT 消息。该消息每 3 小时发送一次。收到该消息后，EntropyService 会再次调用 writeEntropy()函数，将 urandom 设备的内容写到 entropy.dat 中。具体实现代码如下所示。

```
private void scheduleEntropyWriter() {
    mHandler.removeMessages(ENTROPY_WHAT);
    mHandler.sendEmptyMessageDelayed(ENTROPY_WHAT, ENTROPY_WRITE_PERIOD);
}
```

通过上面的分析可知，文件 entropy.dat 保存了 urandom 设备内容的快照（每 3 小时更新一次）。当系统重新启动时，EntropyService 又利用这个文件来增加系统的熵，通过这种方式使随机数的生成更加不可预测。

8.4 生成并管理日志文件

在 Android 5.0 系统中，DropBoxManagerService（DBMS）用于生成和管理系统运行时的一些日志文件。这些日志文件大多记录的是系统或某个应用程序出错时的信息。其中，向 SystemServer 添加 DBMS 的代码如下所示。

```
ServiceManager.addService(Context.DROPBOX_SERVICE,                    //服务名为 dropbox
                          new DropBoxManagerService(context,
                          new File("/data/system/dropbox")));
```

本节将详细讲解 Android 5.0 系统中 DropBoxManagerService 的核心架构知识。

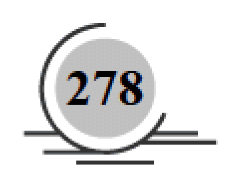
8.4.1 分析 DBMS 构造函数

DBMS 构造函数在文件/frameworks/base/services/java/com/android/server/DropBoxManagerService.java 中实现。

DBMS 构造函数 DropBoxManagerService()的具体实现代码如下所示。

```
public DropBoxManagerService(final Context context, File path) {
    mDropBoxDir = path; //path 指定 dropbox 目录为/data/system/dropbox

    mContext = context;
```



```
    mContentResolver = context.getContentResolver();

    IntentFilter filter = new IntentFilter();
    filter.addAction(Intent.ACTION_DEVICE_STORAGE_LOW);
    filter.addAction(Intent.ACTION_BOOT_COMPLETED);
    //注册一个 Broadcast 监听对象，当系统启动完毕或者设备存储空间不足时，会收到广播
    context.registerReceiver(mReceiver, filter);
    //当 Settings 数据库相应项发生变化时，也需要告知 DBMS 进行相应处理
    mContentResolver.registerContentObserver(
        Settings.Global.CONTENT_URI, true,
        new ContentObserver(new Handler()) {
            @Override
            public void onChange(boolean selfChange) {

                //当 Settings 数据库发生变化时，BroadcastReceiver 的 onReceive()函数
                //将被调用。注意第二个参数为 null
                mReceiver.onReceive(context, (Intent) null);
            }
        });

    mHandler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            if (msg.what == MSG_SEND_BROADCAST) {
                mContext.sendBroadcastAsUser((Intent)msg.obj, UserHandle.OWNER,
                        android.Manifest.permission.READ_LOGS);
            }
        }
    };

}

/** Unregisters broadcast receivers and any other hooks -- for test instances */
public void stop() {
    mContext.unregisterReceiver(mReceiver);
}
```

通过上述代码可知，DBMS 注册一个 BroadcastReceiver 对象，同时会监听 Settings 数据库的变动。其核心逻辑都在此 BroadcastReceiver 的 onReceive()函数中。函数 onReceive()的主要功能是，存储空间不足时需要删除一些旧的日志文件以节省存储空间。函数 onReceive()的具体实现代码如下所示。

```
    public void onReceive(Context context, Intent intent) {
        if (intent != null && Intent.ACTION_BOOT_COMPLETED.equals(intent.getAction())) {
            mBooted = true;
            return;
        }

        mCachedQuotaUptimeMillis = 0;

        new Thread() {
            public void run() {
```

```
                try {
                    init();
                    trimToFit();
                } catch (IOException e) {
                    Slog.e(TAG, "Can't init", e);
                }
            }
        }.start();
    }
};
```

函数 onReceive()会在以下 3 种情况发生时被调用。

- ☑ 当系统启动完毕时，由 BOOT_COMPLETED 广播触发。
- ☑ 当设备存储空间不足时，由 DEVICE_STORAGE_LOW 广播触发。
- ☑ 当 Settings 数据库相应项发生变化时，该函数也会被触发。

8.4.2 添加 dropbox 日志文件

在 Android 5.0 系统中，要想理清一个 Service，最好从它提供的服务开始进行分析。当某个应用程序因为发生异常而崩溃（crash）时，会调用 ActivityManagerService（AMS）的函数 handleApplicationCrash()，此函数在文件/frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中定义。

函数 handleApplicationCrash()的具体实现代码如下所示。

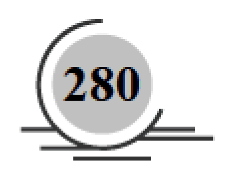
```
public void handleApplicationCrash(IBinder app, ApplicationErrorReport.CrashInfo crashInfo) {
    ProcessRecord r = findAppProcess(app, "Crash");
    final String processName = app == null ? "system_server"
            : (r == null ? "unknown" : r.processName);

    EventLog.writeEvent(EventLogTags.AM_CRASH, Binder.getCallingPid(),
            UserHandle.getUserId(Binder.getCallingUid()), processName,
            r == null ? -1 : r.info.flags,
            crashInfo.exceptionClassName,
            crashInfo.exceptionMessage,
            crashInfo.throwFileName,
            crashInfo.throwLineNumber);
    //调用 addErrorToDropBox()函数，第一个参数是一个字符串，为 crash
    addErrorToDropBox("crash", r, processName, null, null, null, null, null, crashInfo);

    crashApplication(r, crashInfo);
}
```

下面来看函数 addErrorToDropBox()，此函数也在文件 ActivityManagerService.java 中实现，具体实现代码如下所示。

```
public void addErrorToDropBox(String eventType,
        ProcessRecord process, String processName, ActivityRecord activity,
        ActivityRecord parent, String subject,
        final String report, final File logFile,
```



```
        final ApplicationErrorReport.CrashInfo crashInfo) {

    final String dropboxTag = processClass(process) + "_" + eventType;
    final DropBoxManager dbox = (DropBoxManager)
            mContext.getSystemService(Context.DROPBOX_SERVICE);

    if (dbox == null || !dbox.isTagEnabled(dropboxTag)) return;

    final StringBuilder sb = new StringBuilder(1024);
    appendDropBoxProcessHeaders(process, processName, sb);
    if (activity != null) {
        sb.append("Activity: ").append(activity.shortComponentName).append("\n");
    }
    if (parent != null && parent.app != null && parent.app.pid != process.pid) {
        sb.append("Parent-Process: ").append(parent.app.processName).append("\n");
    }
    if (parent != null && parent != activity) {
        sb.append("Parent-Activity: ").append(parent.shortComponentName).append("\n");
    }
    if (subject != null) {
        sb.append("Subject: ").append(subject).append("\n");
    }
    sb.append("Build: ").append(Build.FINGERPRINT).append("\n");
    if (Debug.isDebuggerConnected()) {
        sb.append("Debugger: Connected\n");
    }
    sb.append("\n");

    Thread worker = new Thread("Error dump: " + dropboxTag) {
        @Override
        public void run() {
            if (report != null) {
                sb.append(report);
            }
            if (logFile != null) {
                try {
                    sb.append(FileUtils.readTextFile(logFile, 128 * 1024, "\n\n[[TRUNCATED]]"));
                } catch (IOException e) {
                    Slog.e(TAG, "Error reading " + logFile, e);
                }
            }
            if (crashInfo != null && crashInfo.stackTrace != null) {
                sb.append(crashInfo.stackTrace);
            }

            String setting = Settings.Global.ERROR_LOGCAT_PREFIX + dropboxTag;
            int lines = Settings.Global.getInt(mContext.getContentResolver(), setting, 0);
            if (lines > 0) {
                sb.append("\n");

                InputStreamReader input = null;
```

```
                try {
                    java.lang.Process logcat = new ProcessBuilder("/system/bin/logcat",
                            "-v", "time", "-b", "events", "-b", "system", "-b", "main",
                            "-t", String.valueOf(lines)).redirectErrorStream(true).start();

                    try { logcat.getOutputStream().close(); } catch (IOException e) {}
                    try { logcat.getErrorStream().close(); } catch (IOException e) {}
                    input = new InputStreamReader(logcat.getInputStream());

                    int num;
                    char[] buf = new char[8192];
                    while ((num = input.read(buf)) > 0) sb.append(buf, 0, num);
                } catch (IOException e) {
                    Slog.e(TAG, "Error running logcat", e);
                } finally {
                    if (input != null) try { input.close(); } catch (IOException e) {}
                }
            }

            dbox.addText(dropboxTag, sb.toString());
        }
    };

    if (process == null) {
        worker.run();
    } else {
        worker.start();
    }
}
```

由上述代码可知，函数 addErrorToDropBox()的核心功能是生成日志内容，并调用函数 addText()将内容传给 DBMS。函数 addText()在文件/frameworks/base/core/java/android/os/DropBoxManager.java 中定义。

在 DropBoxManager 类中，函数 addText()的实现代码如下所示。

```
public void addText(String tag, String data) {
    try { mService.add(new Entry(tag, 0, data)); } catch (RemoteException e) {}
}
```

在上述代码中实现了 mService 和 DBMS 的交互。DBMS 对外只提供一个 add()函数实现日志添加工作，而 DBM 提供了 3 个函数，分别是 addText()、addData()、addFile()，以便使用。

DBM 向 DBMS 传递的数据被封装在一个 Entry 中，DBMS 中的函数 add()在文件 DropBoxManagerService.java 中定义，具体实现代码如下所示。

```
public void add(DropBoxManager.Entry entry) {
    File temp = null;
    OutputStream output = null;
```

```
final String tag = entry.getTag();
try {
    int flags = entry.getFlags();
    if ((flags & DropBoxManager.IS_EMPTY) != 0) throw new IllegalArgumentException();

    init();
    if (!isTagEnabled(tag)) return;
    long max = trimToFit();
    long lastTrim = System.currentTimeMillis();

    byte[] buffer = new byte[mBlockSize];
    InputStream input = entry.getInputStream();

    int read = 0;
    while (read < buffer.length) {
        int n = input.read(buffer, read, buffer.length - read);
        if (n <= 0) break;
        read += n;
    }

    temp = new File(mDropBoxDir, "drop" + Thread.currentThread().getId() + ".tmp");
    int bufferSize = mBlockSize;
    if (bufferSize > 4096) bufferSize = 4096;
    if (bufferSize < 512) bufferSize = 512;
    FileOutputStream foutput = new FileOutputStream(temp);
    output = new BufferedOutputStream(foutput, bufferSize);
    if (read == buffer.length && ((flags & DropBoxManager.IS_GZIPPED) == 0)) {
        output = new GZIPOutputStream(output);
        flags = flags | DropBoxManager.IS_GZIPPED;
    }

   do {
        output.write(buffer, 0, read);

        long now = System.currentTimeMillis();
        if (now - lastTrim > 30 * 1000) {
            max = trimToFit();
            lastTrim = now;
        }

        read = input.read(buffer);
        if (read <= 0) {
            FileUtils.sync(foutput);
            output.close();
            output = null;
        } else {
            output.flush();
        }

        long len = temp.length();
        if (len > max) {
```

```
                    Slog.w(TAG, "Dropping: " + tag + " (" + temp.length() + " > " + max + " bytes)");
                    temp.delete();
                    temp = null;
                    break;
                }
            } while (read > 0);

            long time = createEntry(temp, tag, flags);
            temp = null;

            final Intent dropboxIntent = new Intent(DropBoxManager.ACTION_DROPBOX_ENTRY_ADDED);
            dropboxIntent.putExtra(DropBoxManager.EXTRA_TAG, tag);
            dropboxIntent.putExtra(DropBoxManager.EXTRA_TIME, time);
            if (!mBooted) {
                dropboxIntent.addFlags(Intent.FLAG_RECEIVER_REGISTERED_ONLY);
            }
            mHandler.sendMessage(mHandler.obtainMessage(MSG_SEND_BROADCAST, dropboxIntent));
        } catch (IOException e) {
            Slog.e(TAG, "Can't write: " + tag, e);
        } finally {
            try { if (output != null) output.close(); } catch (IOException e) {}
            entry.close();
            if (temp != null) temp.delete();
        }
}
```

从上述代码可知，DBMS 需要考虑每一个日志文件的压缩以节省存储空间。

8.4.3 DBMS 和 settings 数据库

DBMS 的运行依赖一些配置项。其实除了 DBMS 外，SystemServer 中很多服务都依赖相关的配置项。这些配置项都是通过 SettingsProvider 操作 Settings 数据库来设置和查询的。SettingsProvider 是系统中很重要的一个 APK，如果将其删除系统就不能正常启动了。

和系统相关的配置项都在 Settings 数据库的 Secure 表内，具体说明如下所示。

```
//用来判断是否允许记录该 tag 类型的日志文件。默认是允许生成任何 tag 类型的文件
Secure.DROPBOX_TAG_PREFIX+tag: "dropbox:"+tag
//用于控制每个日志文件的存活时间，默认是 3 天。大于 3 天的日志文件就会被删除以节省空间
Secure.DROPBOX_AGE_SECONDS: "dropbox_age_seconds"
//用于控制系统保存的日志文件个数，默认是 1000 个文件
Secure.DROPBOX_MAX_FILES："dropbox_max_files"
//用于控制 dropbox 目录最多占存储空间容量的比例，默认是 10%
Secure.DROPBOX_QUOTA_PERCENT: "dropbox_quota_percent"
//不允许 dropbox 使用的存储空间的比例，默认是 10%，即 dropbox 最多只能使用 90%的空间
Secure.DROPBOX_RESERVE_PERCENT: "dropbox_reserve_percent"
//dropbox 最大能使用的空间大小，默认是 5MB
Secure.DROPBOX_QUOTA_KB:"dropbox_quota_kb"
```

读者可以利用 adb shell 进入/data/data/com.android.providers.settings/databases/目录，然后利用 sqlite3

命令操作 settings.db，通过表 Secure 可以了解相关内容。不过系统中的很多选项在该表中都没有相关设置，因此实际运行时都会使用代码中设置的默认值。

8.5　分析 DiskStatsService

在 Android 5.0 中，DiskStatsService 用于实现内部状态的监控工作。DiskStatsService 通常与 DeviceStroageMonitorService 一起实现与系统内部存储管理和监控有关的服务。DiskStatsService 在文件 /frameworks/base/services/java/com/android/server/DiskStatsService.java 中实现。

文件 DiskStatsService.java 的具体实现代码如下所示。

```
import java.io.File;
import java.io.FileDescriptor;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintWriter;

public class DiskStatsService extends Binder {
    private static final String TAG = "DiskStatsService";

    private final Context mContext;

    public DiskStatsService(Context context) {
        mContext = context;
    }

    @Override
    protected void dump(FileDescriptor fd, PrintWriter pw, String[] args) {
        mContext.enforceCallingOrSelfPermission(android.Manifest.permission.DUMP, TAG);

        byte[] junk = new byte[512];
        for (int i = 0; i < junk.length; i++) junk[i] = (byte) i;

        File tmp = new File(Environment.getDataDirectory(), "system/perftest.tmp");
        FileOutputStream fos = null;
        IOException error = null;

        long before = SystemClock.uptimeMillis();
        try {
            fos = new FileOutputStream(tmp);
            fos.write(junk);
        } catch (IOException e) {
            error = e;
        } finally {
            try { if (fos != null) fos.close(); } catch (IOException e) {}
        }

        long after = SystemClock.uptimeMillis();
```

```
        if (tmp.exists()) tmp.delete();

        if (error != null) {
            pw.print("Test-Error: ");
            pw.println(error.toString());
        } else {
            pw.print("Latency: ");
            pw.print(after - before);
            pw.println("ms [512B Data Write]");
        }

        reportFreeSpace(Environment.getDataDirectory(), "Data", pw);
        reportFreeSpace(Environment.getDownloadCacheDirectory(), "Cache", pw);
        reportFreeSpace(new File("/system"), "System", pw);

    }

    private void reportFreeSpace(File path, String name, PrintWriter pw) {
        try {
            StatFs statfs = new StatFs(path.getPath());
            long bsize = statfs.getBlockSize();
            long avail = statfs.getAvailableBlocks();
            long total = statfs.getBlockCount();
            if (bsize <= 0 || total <= 0) {
                throw new IllegalArgumentException(
                        "Invalid stat: bsize=" + bsize + " avail=" + avail + " total=" + total);
            }

            pw.print(name);
            pw.print("-Free: ");
            pw.print(avail * bsize / 1024);
            pw.print("K / ");
            pw.print(total * bsize / 1024);
            pw.print("K total = ");
            pw.print(avail * 100 / total);
            pw.println("% free");
        } catch (IllegalArgumentException e) {
            pw.print(name);
            pw.print("-Error: ");
            pw.println(e.toString());
            return;
        }
    }
}
```

从上述代码可以看出，虽然 DiskStatsService 从 Binder 中派生，但是并没有实现任何接口，即 DiskStatsService 没有任何可调用的业务函数。但是在系统中为什么会存在这样的服务呢？要想解决这个问题，需要先了解系统中的命令 dumpsys，此命令用于打印系统中指定服务的信息，在文件/frameworks/native/cmds/dumpsys/dumpsys.cpp 中定义。



文件 dumpsys.cpp 的具体实现代码如下所示。

```
#define LOG_TAG "dumpsys"

#include <utils/Log.h>
#include <binder/Parcel.h>
#include <binder/ProcessState.h>
#include <binder/IServiceManager.h>
#include <utils/TextOutput.h>
#include <utils/Vector.h>

#include <getopt.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/time.h>

using namespace android;

static int sort_func(const String16* lhs, const String16* rhs)
{
    return lhs->compare(*rhs);
}

int main(int argc, char* const argv[])
{
    signal(SIGPIPE, SIG_IGN);
    sp<IServiceManager> sm = defaultServiceManager();
    fflush(stdout);
    if (sm == NULL) {
        ALOGE("Unable to get default service manager!");
        aerr << "dumpsys: Unable to get default service manager!" << endl;
        return 20;
    }

    Vector<String16> services;
    Vector<String16> args;
    if (argc == 1) {
        services = sm->listServices();
        services.sort(sort_func);
        args.add(String16("-a"));
    } else {
        services.add(String16(argv[1]));
        for (int i=2; i<argc; i++) {
            args.add(String16(argv[i]));
        }
    }

    const size_t N = services.size();
```

```
    if (N > 1) {
        aout << "Currently running services:" << endl;

        for (size_t i=0; i<N; i++) {
            sp<IBinder> service = sm->checkService(services[i]);
            if (service != NULL) {
                aout << "   " << services[i] << endl;
            }
        }
    }

    for (size_t i=0; i<N; i++) {
        sp<IBinder> service = sm->checkService(services[i]);
        if (service != NULL) {
            if (N > 1) {
                aout << "------------------------------------------------------------"
                        "-------------------" << endl;
                aout << "DUMP OF SERVICE " << services[i] << ":" << endl;
            }
            int err = service->dump(STDOUT_FILENO, args);
            if (err != 0) {
                aerr << "Error dumping service info: (" << strerror(err)
                        << ") " << services[i] << endl;
            }
        } else {
            aerr << "Can't find service: " << services[i] << endl;
        }
    }

    return 0;
}
```

通过上述代码可知，dumpsys 通过 Binder 调用某个 Service 的 dump()函数。上述代码的具体实现流程如下所示。

（1）先获取与 ServiceManager 进程通信的 BpServiceManager 对象。

（2）如果输入参数个数为 1，则先查询在 SM 中注册的所有 Service。

（3）将 Service 排序。

（4）指定查询某个 Service。

（5）保存剩余参数，以后可以传给 Service 的 dump()函数。

（6）通过 Binder 调用该 Service 的 dump()函数，将 args 也传给 dump()函数。

接下来看文件 DiskStatsService.java 中的函数 dump()，具体实现代码如下所示。

```
protected void dump(FileDescriptor fd, PrintWriter pw, String[] args) {
    mContext.enforceCallingOrSelfPermission(android.Manifest.permission.DUMP, TAG);

    byte[] junk = new byte[512];
    for (int i = 0; i < junk.length; i++) junk[i] = (byte) i;

    File tmp = new File(Environment.getDataDirectory(), "system/perftest.tmp");
```

```
        FileOutputStream fos = null;
        IOException error = null;

        long before = SystemClock.uptimeMillis();
        try {
            fos = new FileOutputStream(tmp);
            fos.write(junk);
        } catch (IOException e) {
            error = e;
        } finally {
            try { if (fos != null) fos.close(); } catch (IOException e) {}
        }

        long after = SystemClock.uptimeMillis();
        if (tmp.exists()) tmp.delete();

        if (error != null) {
            pw.print("Test-Error: ");
            pw.println(error.toString());
        } else {
            pw.print("Latency: ");
            pw.print(after - before);
            pw.println("ms [512B Data Write]");
        }

        reportFreeSpace(Environment.getDataDirectory(), "Data", pw);
        reportFreeSpace(Environment.getDownloadCacheDirectory(), "Cache", pw);
        reportFreeSpace(new File("/system"), "System", pw);

}
```

从上述代码可知，DiskStatsService 没有实现任何业务接口，只是为了调试而存在。

8.6　监测系统内部存储空间的状态

在 Android 5.0 中，DeviceStorageManagerService（DSMS）用于监测系统内部存储空间的状态，添加该服务的代码如下所示。

```
//DSMS 的服务名为 devicestoragemonitor
ServiceManager.addService(DeviceStorageMonitorService.SERVICE,
new DeviceStorageMonitorService(context));
```

本节将详细讲解 Android 5.0 中 DeviceStorageManagerService 的具体架构知识。

8.6.1　构造函数

DSMS 的构造函数在文件/frameworks/base/services/java/com/android/server/DeviceStorageMonitorService.java

中实现。

函数 DeviceStorageMonitorService()的具体实现代码如下所示。

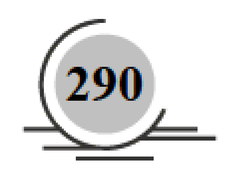
```
public DeviceStorageMonitorService(Context context) {
    mLastReportedFreeMemTime = 0;
    mContext = context;
    mContentResolver = mContext.getContentResolver();
    mDataFileStats = new StatFs(DATA_PATH);//获取 data 分区的信息
    mSystemFileStats = new StatFs(SYSTEM_PATH);//获取 system 分区的信息
    mCacheFileStats = new StatFs(CACHE_PATH);//获取 cache 分区的信息
    //获得 data 分区的总大小
    mTotalMemory = ((long)mDataFileStats.getBlockCount() *
                    mDataFileStats.getBlockSize())/100L;
    /*
    创建 3 个 Intent，分别用于通知存储空间不足、存储空间恢复正常和存储空间满。
    由于设置了 REGISTERED_ONLY_BEFORE_BOOT 标志，这 3 个 Intent 广播只能由
    系统服务接收
    */
    mStorageLowIntent = new Intent(Intent.ACTION_DEVICE_STORAGE_LOW);
    mStorageLowIntent.addFlags(
                    Intent.FLAG_RECEIVER_REGISTERED_ONLY_BEFORE_BOOT);
    mStorageOkIntent = new Intent(Intent.ACTION_DEVICE_STORAGE_OK);
    mStorageOkIntent.addFlags(
                Intent.FLAG_RECEIVER_REGISTERED_ONLY_BEFORE_BOOT);
    mStorageFullIntent = new Intent(Intent.ACTION_DEVICE_STORAGE_FULL);
    mStorageFullIntent.addFlags(
                Intent.FLAG_RECEIVER_REGISTERED_ONLY_BEFORE_BOOT);
    mStorageNotFullIntent = new
                Intent(Intent.ACTION_DEVICE_STORAGE_NOT_FULL);
    mStorageNotFullIntent.addFlags(
                Intent.FLAG_RECEIVER_REGISTERED_ONLY_BEFORE_BOOT);

    //查询 Settings 数据库中 sys_storage_threshold_percentage 的值，默认是 10，
    //即当/data 空间只剩 10%时，认为空间不足
    mMemLowThreshold = getMemThreshold();
    //查询 Settings 数据库中 sys_storage_full_threshold_bytes 的值，默认是 1MB，
    //即当 data 分区只剩 1MB 时，就认为空间已满，剩下的 1MB 空间保留给系统自用
    mMemFullThreshold = getMemFullThreshold();
    //检查内存
    checkMemory(true);
}
```

8.6.2 内存检查

再来看内存检查函数 checkMemory()，此函数也是在文件 DeviceStorageMonitorService.java 中定义的，具体实现代码如下所示。

```
private final void checkMemory(boolean checkCache) {
    if(mClearingCache) {
        if(localLOGV) Slog.i(TAG, "Thread already running just skip");
```



```
            long diffTime = System.currentTimeMillis() - mThreadStartTime;
            if(diffTime > (10*60*1000)) {
                Slog.w(TAG, "Thread that clears cache file seems to run for ever");
            }
        } else {
            restatDataDir();
            if (localLOGV)  Slog.v(TAG, "freeMemory="+mFreeMem);
            if (mFreeMem < mMemLowThreshold) {
                if (checkCache) {
                    if (mFreeMem < mMemCacheStartTrimThreshold) {
                        if ((mFreeMemAfterLastCacheClear-mFreeMem)
                                >= ((mMemLowThreshold-mMemCacheStartTrimThreshold)/4)) {
                            mThreadStartTime = System.currentTimeMillis();
                            mClearSucceeded = false;
                            clearCache();
                        }
                    }
                } else {
                    mFreeMemAfterLastCacheClear = mFreeMem;
                    if (!mLowMemFlag) {
                        Slog.i(TAG, "Running low on memory. Sending notification");
                        sendNotification();
                        mLowMemFlag = true;
                    } else {
                        if (localLOGV) Slog.v(TAG, "Running low on memory " +
                                "notification already sent. do nothing");
                    }
                }
            } else {
                mFreeMemAfterLastCacheClear = mFreeMem;
                if (mLowMemFlag) {
                    Slog.i(TAG, "Memory available. Cancelling notification");
                    cancelNotification();
                    mLowMemFlag = false;
                }
            }
            if (mFreeMem < mMemFullThreshold) {
                if (!mMemFullFlag) {
                    sendFullNotification();
                    mMemFullFlag = true;
                }
            } else {
                if (mMemFullFlag) {
                    cancelFullNotification();
                    mMemFullFlag = false;
                }
            }
        }
        if(localLOGV) Slog.i(TAG, "Posting Message again");
        postCheckMemoryMsg(true, DEFAULT_CHECK_INTERVAL);
}
```

当空间不足时，DSMS 会先使用函数 clearCache()进行处理，在此函数内部会与 PackageManager-Service（简称 PKMS）进行交互。函数 clearCache()在文件 DeviceStorageManagerService.java 中定义，具体实现代码如下所示。

```
private final void clearCache() {
    if (mClearCacheObserver == null) {
        mClearCacheObserver = new CachePackageDataObserver();
    }
    mClearingCache = true;
    try {
        if (localLOGV) Slog.i(TAG, "Clearing cache");
        IPackageManager.Stub.asInterface(ServiceManager.getService("package")).
                freeStorageAndNotify(mMemCacheTrimToThreshold, mClearCacheObserver);
    } catch (RemoteException e) {
        Slog.w(TAG, "Failed to get handle for PackageManger Exception: "+e);
        mClearingCache = false;
        mClearSucceeded = false;
    }
}
```

CachePackageDataObserver 是 DSMS 定义的内部类，其中的函数 onRemoveCompleted()用于重新发送消息，让 DSMS 再检测一次存储空间。函数 DeviceStorageManagerService()并没有重载 dump()函数。

8.7 分析实现性能统计

在 Android 5.0 的源码中，SamplingProfilerService 的功能是实现性能统计工作。在 Android 应用中，添加 SamplingProfilerService 服务的实现代码如下所示。

```
ServiceManager.addService("samplingprofiler",//服务名
                                        new SamplingProfilerService(context));
```

本节将详细分析 Android 5.0 中 SamplingProfilerService 的核心架构知识。

8.7.1 构造函数

SamplingProfilerService 的构造函数在文件/frameworks/base/services/java/com/android/server/SamplingProfilerService.java 中实现。

在文件 SamplingProfilerService.java 中，函数 SamplingProfilerService()的具体实现代码如下所示。

```
public SamplingProfilerService(Context context) {
        //注册一个 CotentObserver，用于监测 Settings 数据库的变化
        registerSettingObserver(context);
        startWorking(context);//startWorking 函数
}
```

上述代码的核心是函数 startWorking()，此函数在文件 SamplingProfilerService.java 中定义，具体实



现代码如下所示。

```
private void startWorking(Context context) {
    if (LOCAL_LOGV) Slog.v(TAG, "starting SamplingProfilerService!");

    final DropBoxManager dropbox =
            (DropBoxManager) context.getSystemService(Context.DROPBOX_SERVICE);

    File[] snapshotFiles = new File(SNAPSHOT_DIR).listFiles();
    for (int i = 0; snapshotFiles != null && i < snapshotFiles.length; i++) {
        handleSnapshotFile(snapshotFiles[i], dropbox);
    }

    snapshotObserver = new FileObserver(SNAPSHOT_DIR, FileObserver.ATTRIB) {
        @Override
        public void onEvent(int event, String path) {
            handleSnapshotFile(new File(SNAPSHOT_DIR, path), dropbox);
        }
    };
    snapshotObserver.startWatching();

    if (LOCAL_LOGV) Slog.v(TAG, "SamplingProfilerService activated");
}
```

通过上述代码可知，SamplingProfilerService 本身并不提供性能统计的功能。统计功能是通过类 SamplingProfilerIntegration 实现的，这个类封装了一个 SamplingProfiler（由 Dalvik 虚拟机提供）对象，并提供了方便利用的函数进行性能统计。

8.7.2　进行性能统计

通过使用 SamplingProfilerIntegration 可以进行性能统计。在 Andorid 系统中有很多重要进程都需要对性能进行分析，例如 Zygote，其相关代码在文件/frameworks/base/core/java/com/android/internal/os/ZygoteInit.java 中实现。

在文件 ZygoteInit.java 中，和性能分析相关的代码如下所示。

```
public static void main(String argv[]) {
    try {
        SamplingProfilerIntegration.start();

        registerZygoteSocket();
        EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_START,
            SystemClock.uptimeMillis());
        preload();
        EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_END,
            SystemClock.uptimeMillis());

        SamplingProfilerIntegration.writeZygoteSnapshot();

        gc();
```

```
            if (argv.length != 2) {
                throw new RuntimeException(argv[0] + USAGE_STRING);
            }

            if (argv[1].equals("start-system-server")) {
                startSystemServer();
            } else if (!argv[1].equals("")) {
                throw new RuntimeException(argv[0] + USAGE_STRING);
            }

            Log.i(TAG, "Accepting command socket connections");

            if (ZYGOTE_FORK_MODE) {
                runForkMode();
            } else {
                runSelectLoopMode();
            }

            closeServerSocket();
        } catch (MethodAndArgsCaller caller) {
            caller.run();
        } catch (RuntimeException ex) {
            Log.e(TAG, "Zygote died with exception", ex);
            closeServerSocket();
            throw ex;
        }
}
```

在上述代码中，函数 start()在文件/frameworks/base/core/java/com/android/internal/os/SamplingProfilerIntegration.java 中实现。

函数 start()的具体实现代码如下所示。

```
public static void start() {
    if (!enabled) {         //判断是否开启性能统计
        return;
    }
    if (samplingProfiler != null) {
        Log.e(TAG, "SamplingProfilerIntegration already started at " + new Date(startMillis));
        return;
    }

    ThreadGroup group = Thread.currentThread().getThreadGroup();
    //创建一个 Dalvik 的 SamplingProfiler
    SamplingProfiler.ThreadSet threadSet = SamplingProfiler.newThreadGroupTheadSet(group);
    samplingProfiler = new SamplingProfiler(samplingProfilerDepth, threadSet);
    //启动统计
    samplingProfiler.start(samplingProfilerMilliseconds);
    startMillis = System.currentTimeMillis();
}
```

在上述代码中，使用该类的 static 语句来判断启动性能统计的 enable 变量由谁控制。在文件 SamplingProfilerIntegration.java 中，static 语句的实现代码如下所示。

```
static {
    samplingProfilerMilliseconds = SystemProperties.getInt("persist.sys.profiler_ms", 0);
    samplingProfilerDepth = SystemProperties.getInt("persist.sys.profiler_depth", 4);
    if (samplingProfilerMilliseconds > 0) {
        File dir = new File(SNAPSHOT_DIR);
        dir.mkdirs();
        dir.setWritable(true, false);
        dir.setExecutable(true, false);
        if (dir.isDirectory()) {
            snapshotWriter = Executors.newSingleThreadExecutor(new ThreadFactory() {
                    public Thread newThread(Runnable r) {
                        return new Thread(r, TAG);
                    }
                });
            enabled = true;
            Log.i(TAG, "Profiling enabled. Sampling interval ms: "
                    + samplingProfilerMilliseconds);
        } else {
            snapshotWriter = null;
            enabled = true;
            Log.w(TAG, "Profiling setup failed. Could not create " + SNAPSHOT_DIR);
        }
    } else {
        snapshotWriter = null;
        enabled = false;
        Log.i(TAG, "Profiling disabled.");
    }
}
```

由上述代码可知，enable 的控制在 static 语句中实现，这表明要使用性能统计，就必须重新启动要统计的进程。

8.7.3　输出统计文件

当启动性能统计后，需要输出统计文件，此功能由函数 writeZygoteSnapshot()实现。在文件 SamplingProfilerIntegration.java 中，函数 writeZygoteSnapshot()的具体实现代码如下所示。

```
public static void writeZygoteSnapshot() {
    if (!enabled) {
        return;
    }
    writeSnapshotFile("zygote", null);
    samplingProfiler.shutdown();
    samplingProfiler = null;
    startMillis = 0;
}
```

在上述代码中，调用了 writeSnapshotFile()函数，其第一个参数为 zygote，用于表示进程名。writeSnapshotFile()函数比较简单，功能就是在 shots 目录下生成一个统计文件，统计文件的名称由两部分组成，合起来就是“进程名_开始 性能统计的时刻.snapshot”。另外，writeSnapshotfile()内部会调用 generateSnapshotHeader()函数在该统计文件的头部写一些特定的信息，例如版本号、编译信息等。在文件 SamplingProfilerIntegration.java 中，函数 writeSnapshotFile()的具体实现代码如下所示。

```
private static void writeSnapshotFile(String processName, PackageInfo packageInfo) {
    if (!enabled) {
        return;
    }
    samplingProfiler.stop();
    String name = processName.replaceAll(":", ".");
    String path = SNAPSHOT_DIR + "/" + name + "-" + startMillis + ".snapshot";
    long start = System.currentTimeMillis();
    OutputStream outputStream = null;
    try {
        outputStream = new BufferedOutputStream(new FileOutputStream(path));
        PrintStream out = new PrintStream(outputStream);
        generateSnapshotHeader(name, packageInfo, out);
        if (out.checkError()) {
            throw new IOException();
        }
        BinaryHprofWriter.write(samplingProfiler.getHprofData(), outputStream);
    } catch (IOException e) {
        Log.e(TAG, "Error writing snapshot to " + path, e);
        return;
    } finally {
        IoUtils.closeQuietly(outputStream);
    }
    new File(path).setReadable(true, false);

    long elapsed = System.currentTimeMillis() - start;
    Log.i(TAG, "Wrote snapshot " + path + " in " + elapsed + "ms.");
    samplingProfiler.start(samplingProfilerMilliseconds);
}
```

SamplingProfilerIntegration 的核心是类 SamplingProfiler，这个类定义在文件 libcore/dalvik/src/main/java/dalvik/system/profiler/SamplingProfiler.java 中。

文件 SamplingProfiler.java 的具体实现代码如下所示。

```
public final class SamplingProfiler {
    private final Map<HprofData.StackTrace, int[]> stackTraces
            = new HashMap<HprofData.StackTrace, int[]>();
    private final HprofData hprofData = new HprofData(stackTraces);
    private final Timer timer = new Timer("SamplingProfiler", true);
    private Sampler sampler;
    private final int depth;
    private final ThreadSet threadSet;
    private int nextThreadId = 200001;
```

```
private int nextStackTraceId = 300001;
private int nextObjectId = 1;
private Thread[] currentThreads = new Thread[0];
private final Map<Thread, Integer> threadIds = new HashMap<Thread, Integer>();
private final HprofData.StackTrace mutableStackTrace = new HprofData.StackTrace();
private final ThreadSampler threadSampler;
public SamplingProfiler(int depth, ThreadSet threadSet) {
    this.depth = depth;
    this.threadSet = threadSet;
    this.threadSampler = findDefaultThreadSampler();
    threadSampler.setDepth(depth);
    hprofData.setFlags(BinaryHprof.ControlSettings.CPU_SAMPLING.bitmask);
    hprofData.setDepth(depth);
}

private static ThreadSampler findDefaultThreadSampler() {
    if ("Dalvik Core Library".equals(System.getProperty("java.specification.name"))) {
        String className = "dalvik.system.profiler.DalvikThreadSampler";
        try {
            return (ThreadSampler) Class.forName(className).newInstance();
        } catch (Exception e) {
            System.out.println("Problem creating " + className + ": " + e);
        }
    }
    return new PortableThreadSampler();
}

/**
 * A ThreadSet specifies the set of threads to sample
 */
public static interface ThreadSet {
    public Thread[] threads();
}
public static ThreadSet newArrayThreadSet(Thread... threads) {
    return new ArrayThreadSet(threads);
}
private static class ArrayThreadSet implements ThreadSet {
    private final Thread[] threads;
    public ArrayThreadSet(Thread... threads) {
        if (threads == null) {
            throw new NullPointerException("threads == null");
        }
        this.threads = threads;
    }
    public Thread[] threads() {
        return threads;
    }
}

public static ThreadSet newThreadGroupTheadSet(ThreadGroup threadGroup) {
    return new ThreadGroupThreadSet(threadGroup);
```

```
}
private static class ThreadGroupThreadSet implements ThreadSet {
    private final ThreadGroup threadGroup;
    private Thread[] threads;
    private int lastThread;

    public ThreadGroupThreadSet(ThreadGroup threadGroup) {
        if (threadGroup == null) {
            throw new NullPointerException("threadGroup == null");
        }
        this.threadGroup = threadGroup;
        resize();
    }

    private void resize() {
        int count = threadGroup.activeCount();
        threads = new Thread[count*2];
        lastThread = 0;
    }

    public Thread[] threads() {
        int threadCount;
        while (true) {
            threadCount = threadGroup.enumerate(threads);
            if (threadCount == threads.length) {
                resize();
            } else {
                break;
            }
        }
        if (threadCount < lastThread) {
            Arrays.fill(threads, threadCount, lastThread, null);
        }
        lastThread = threadCount;
        return threads;
    }
}
public void start(int interval) {
    if (interval < 1) {
        throw new IllegalArgumentException("interval < 1");
    }
    if (sampler != null) {
        throw new IllegalStateException("profiling already started");
    }
    sampler = new Sampler();
    hprofData.setStartMillis(System.currentTimeMillis());
    timer.scheduleAtFixedRate(sampler, 0, interval);
}

public void stop() {
    if (sampler == null) {
```

```
                return;
            }
            synchronized(sampler) {
                sampler.stop = true;
                while (!sampler.stopped) {
                    try {
                        sampler.wait();
                    } catch (InterruptedException ignored) {
                    }
                }
            }
            sampler = null;
        }
        public void shutdown() {
            stop();
            timer.cancel();
        }
        public HprofData getHprofData() {
            if (sampler != null) {
                throw new IllegalStateException("cannot access hprof data while sampling");
            }
            return hprofData;
        }
        private class Sampler extends TimerTask {

            private boolean stop;
            private boolean stopped;

            private Thread timerThread;

            public void run() {
                synchronized(this) {
                    if (stop) {
                        cancel();
                        stopped = true;
                        notifyAll();
                        return;
                    }
                }

                if (timerThread == null) {
                    timerThread = Thread.currentThread();
                }
                Thread[] newThreads = threadSet.threads();
                if (!Arrays.equals(currentThreads, newThreads)) {
                    updateThreadHistory(currentThreads, newThreads);
                    currentThreads = newThreads.clone();
                }

                for (Thread thread : currentThreads) {
                    if (thread == null) {
```

```
                break;
            }
            if (thread == timerThread) {
                continue;
            }

            StackTraceElement[] stackFrames = threadSampler.getStackTrace(thread);
            if (stackFrames == null) {
                continue;
            }
            recordStackTrace(thread, stackFrames);
        }
    }
    private void recordStackTrace(Thread thread, StackTraceElement[] stackFrames) {
        Integer threadId = threadIds.get(thread);
        if (threadId == null) {
            throw new IllegalArgumentException("Unknown thread " + thread);
        }
        mutableStackTrace.threadId = threadId;
        mutableStackTrace.stackFrames = stackFrames;

        int[] countCell = stackTraces.get(mutableStackTrace);
        if (countCell == null) {
            countCell = new int[1];
            StackTraceElement[] stackFramesCopy = stackFrames.clone();
            HprofData.StackTrace stackTrace
                    = new HprofData.StackTrace(nextStackTraceId++, threadId, stackFramesCopy);
            hprofData.addStackTrace(stackTrace, countCell);
        }
        countCell[0]++;
    }

    private void updateThreadHistory(Thread[] oldThreads, Thread[] newThreads) {
        Set<Thread> n = new HashSet<Thread>(Arrays.asList(newThreads));
        Set<Thread> o = new HashSet<Thread>(Arrays.asList(oldThreads));

        // added = new-old
        Set<Thread> added = new HashSet<Thread>(n);
        added.removeAll(o);

        // removed = old-new
        Set<Thread> removed = new HashSet<Thread>(o);
        removed.removeAll(n);

        for (Thread thread : added) {
            if (thread == null) {
                continue;
            }
            if (thread == timerThread) {
                continue;
            }
```

```
                addStartThread(thread);
            }
            for (Thread thread : removed) {
                if (thread == null) {
                    continue;
                }
                if (thread == timerThread) {
                    continue;
                }
                addEndThread(thread);
            }
        }
        private void addStartThread(Thread thread) {
            if (thread == null) {
                throw new NullPointerException("thread == null");
            }
            int threadId = nextThreadId++;
            Integer old = threadIds.put(thread, threadId);
            if (old != null) {
                throw new IllegalArgumentException("Thread already registered as " + old);
            }

            String threadName = thread.getName();
            ThreadGroup group = thread.getThreadGroup();
            String groupName = group == null ? null : group.getName();
            ThreadGroup parentGroup = group == null ? null : group.getParent();
            String parentGroupName = parentGroup == null ? null : parentGroup.getName();

            HprofData.ThreadEvent event
                    = HprofData.ThreadEvent.start(nextObjectId++, threadId,
                                                  threadName, groupName, parentGroupName);
            hprofData.addThreadEvent(event);
        }

        /**
         * Record that a thread has disappeared
         */
        private void addEndThread(Thread thread) {
            if (thread == null) {
                throw new NullPointerException("thread == null");
            }
            Integer threadId = threadIds.remove(thread);
            if (threadId == null) {
                throw new IllegalArgumentException("Unknown thread " + thread);
            }
            HprofData.ThreadEvent event = HprofData.ThreadEvent.end(threadId);
            hprofData.addThreadEvent(event);
        }
    }
}
```

8.8 剪贴板服务

在 Android 5.0 的源码中，类 content.ClipboardManager 继承自类 text.ClipboardManager，早期的剪贴功能只支持文本。ClipboardManager 由剪贴板服务的客户端使用，在 SDK 中有相应的文档说明。目前，Android 系统中的剪贴板支持 3 种类型的数据（Text、Intent 以及 URL 列表）。

本节将通过一个示例分析 CBS 剪贴板服务的知识，该示例来源于 Android SDK 提供的一段示例代码，路径为/sdk/samples/android-17/。

8.8.1 复制数据到剪贴板

在 Android SDK 的实例源码中，截取如下与复制操作相关的代码。

```
//获取能与 CBS 交互的 ClipboardManager 对象
ClipboardManager clipboard = (ClipboardManager)
                        getSystemService(Context.CLIPBOARD_SERVICE);
//调用 setPrimaryClip()函数，参数是 ClipData.newUri()函数的返回值
clipboard.setPrimaryClip(ClipData.newUri(
                        getContentResolver(),"Note",noteUri));
```

在上述代码中，ClipData 中的 newUri 是一个 static 函数，用于返回一个存储 URI 数据类型的 ClipData，ClipData 对象装载的就是可保存在剪贴板中的数据。函数 newUri()在文件/frameworks/base/core/java/android/content/ClipData.java 中实现。

函数 newUri()的具体实现代码如下所示。

```
static public ClipData newUri(ContentResolver resolver, CharSequence label,
        Uri uri) {
    Item item = new Item(uri);
    String[] mimeTypes = null;
    if ("content".equals(uri.getScheme())) {
        String realType = resolver.getType(uri);
        mimeTypes = resolver.getStreamTypes(uri, "*/*");
        if (mimeTypes == null) {
            if (realType != null) {
                mimeTypes = new String[] { realType, ClipDescription.MIMETYPE_TEXT_URILIST };
            }
        } else {
            String[] tmp = new String[mimeTypes.length + (realType != null ? 2 : 1)];
            int i = 0;
            if (realType != null) {
                tmp[0] = realType;
                i++;
            }
            System.arraycopy(mimeTypes, 0, tmp, i, mimeTypes.length);
            tmp[i + mimeTypes.length] = ClipDescription.MIMETYPE_TEXT_URILIST;
            mimeTypes = tmp;
```

```
            }
        }
        if (mimeTypes == null) {
            mimeTypes = MIMETYPES_TEXT_URILIST;
        }
        return new ClipData(label, mimeTypes, item);
}
```

函数 newUri()的主要功能是获得 uri 所指向的数据的数据类型。对于使用剪贴板服务的程序来说，了解剪贴板中数据的数据类型相当重要，因为这样可以判断自己能否处理这种类型的数据。在上述代码中，uri 指向数据的位置，这和 PC 上文件的存储位置类似，例如 c:/dfp。MIME 则表示该数据的数据类型。在 Windows 平台上是采用后缀名来表示文件类型的，前面提到的 C 盘下的 DFP 文件，后缀是.wav，表示该文件是一个 WAV 格式音频。对于剪贴板来说，数据源由 uri 指定，数据类型由 MIME 表示，两者缺一不可。

当获得一个 ClipData 后，会调用函数 setPrimaryClip()，功能是将数据传递到 CBS。函数 setPrimaryClip()在文件/frameworks/base/core/java/android/content/ClipboardManager.java 中实现。

函数 setPrimaryClip()的具体实现代码如下所示。

```
public void setPrimaryClip(ClipData clip) {
try {
            //跨 Binder 调用，先要把参数打包。有兴趣的读者可以查看 writeToParcel()函数
            getService().setPrimaryClip(clip);
      } catch (RemoteException e) {
   }
}
```

通过 Binder 发送 setPrimaryClip 请求后，由 CBS 完成实际功能。在文件/frameworks/base/services/java/com/android/server/ClipboardService.java 中，函数 setPrimaryClip()的具体实现代码如下所示。

```
public void setPrimaryClip(ClipData clip) {
    synchronized (this) {
        if (clip != null && clip.getItemCount() <= 0) {
            throw new IllegalArgumentException("No items");
        }
        checkDataOwnerLocked(clip, Binder.getCallingUid());
        clearActiveOwnersLocked();
        PerUserClipboard clipboard = getClipboard();
        clipboard.primaryClip = clip;
        final int n = clipboard.primaryClipListeners.beginBroadcast();
        for (int i = 0; i < n; i++) {
            try {
                clipboard.primaryClipListeners.getBroadcastItem(i).dispatchPrimaryClipChanged();
            } catch (RemoteException e) {

            }
        }
        clipboard.primaryClipListeners.finishBroadcast();
    }
}
```

8.8.2 从剪贴板粘贴数据

请读者继续看 SDK 安装包中的示例，演示代码如下。

```
final void performPaste() {
    //获取 ClipboardManager 对象
    ClipboardManager clipboard = (ClipboardManager)
            getSystemService(Context.CLIPBOARD_SERVICE);

    //获取 ContentResolver 对象
    ContentResolver cr = getContentResolver();
    //从剪贴板中取出 ClipData
    ClipData clip = clipboard.getPrimaryClip();
    if (clip != null) {
        String text=null;
        String title=null;
        //取剪贴板 ClipData 中的第一项 Item
        ClipData.Item item = clip.getItemAt(0);
        /*
        取出 Item 中所包含的 uri
        */
        Uri uri = item.getUri();
        Cursor orig = cr.query(uri,PROJECTION, null, null,null);
           ……//查询数据库并获取信息
           orig.close();
            }
        }
        if (text == null) {
            //如果 paste 方不了解 ClipData 中的数据类型，可调用 coerceToText()函数，强制得到文本类型的
数据
           text = item.coerceToText(this).toString();//强制为文本
    }
```

在上述代码中用到了函数 getPrimaryClip()，此函数在文件 ClipboardManager.java 中定义，具体实现代码如下所示。

```
public ClipData getPrimaryClip() {
    try {
        return getService().getPrimaryClip(mContext.getPackageName());
    } catch (RemoteException e) {
        return null;
    }
}
```

在文件 ClipboardManagerService.java 中，函数的具体实现代码如下所示。

```
public ClipData getPrimaryClip(String pkg) {
      synchronized (this) {
          //赋予该 pkg 相应的权限，后文再作分析
          addActiveOwnerLocked(Binder.getCallingUid(), pkg);
```

```
            return mPrimaryClip;//返回 ClipData 给客户端
        }
}
```

在上述代码中，函数 coerceToTe()在 paste 方不了解 ClipData 中数据类型的情况下，可以强制得到文本类型的数据。

再看文件 ClipData.java，在其中定义了 coerceToText，具体实现代码如下所示。

```
public CharSequence coerceToText(Context context) {
    CharSequence text = getText();
    if (text != null) {
        return text;
    }

    Uri uri = getUri();
    if (uri != null) {

        FileInputStream stream = null;
        try {
            AssetFileDescriptor descr = context.getContentResolver()
                        .openTypedAssetFileDescriptor(uri, "text/*", null);
            stream = descr.createInputStream();
            InputStreamReader reader = new InputStreamReader(stream, "UTF-8");

            StringBuilder builder = new StringBuilder(128);
            char[] buffer = new char[8192];
            int len;
            while ((len=reader.read(buffer)) > 0) {
                builder.append(buffer, 0, len);
            }
            return builder.toString();

        } catch (FileNotFoundException e) {

        } catch (IOException e) {
            Log.w("ClippedData", "Failure loading text", e);
            return e.toString();

        } finally {
            if (stream != null) {
                try {
                    stream.close();
                } catch (IOException e) {
                }
            }
        }

        return uri.toString();
    }

    Intent intent = getIntent();
```

```
    if (intent != null) {
        return intent.toUri(Intent.URI_INTENT_SCHEME);
    }

    return "";
}
```

由上述实现代码可知，针对 URI 类型的数据，函数 coerceToText()实现了处理功能。当然，还需要提供该 URI 的 ContentProvider 实现相应的函数。

8.8.3 管理 CBS 中的权限

在 Android 5.0 源码中，CBS 和权限管理相关的函数调用如下所示。

```
//copy 方设置 ClipData 在 CBS 的 setPrimaryClip()函数中进行
checkDataOwnerLocked(clip, Binder.getCallingUid());
clearActiveOwnersLocked();
//paste 方获取 ClipData 在 CBS 的 getPrimaryClip()函数中进行
addActiveOwnerLocked(Binder.getCallingUid(), pkg);
```

（1）URI 权限管理介绍

Android 系统的权限管理中有一类是专门针对 URI 的，先来看一个示例，该示例来自 package/providers/ContactsProvider，在其对应的文件 AndroidManifest.xml 中有如下声明代码。

```
<prvider android:name="ContactsProvider2"
          ...
          android:readPermission="android.permission.READ_CONTACTS"
          android:writePermission="android.permission.WRITE_CONTACTS">
          ...
          <grant-uri-permission android:pathPattern=".*" />
</provider>
```

在上述代码中声明了一个名为 ContactsProvider2 的 ContentProvider，并定义了几个权限声明，具体说明如下所示。

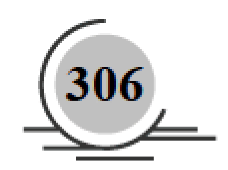
- ☑ readPermission：要求调用 query()函数的客户端必须声明一个 use-permission 为 READ_CONTACTS 的权限。
- ☑ writePermission：要求调用 update()或 insert()函数的客户端必须声明一个 use-permission 为 WRITE_CONTACTS 的权限。
- ☑ grant-uri-permission：和授权有关。

Contacts 和 ContactProvider 这两个 APP 都是由系统提供的程序，而且两者的关系十分紧密，所以 Contacts 一定会声明 use_Permission 为 READ_CONTACTS 和 WRITE_CONTACT 的权限。这样，Contacts 就可以通过 ContactsProvider 来查询或更新数据库了。

（2）分析函数 checkDataOwnerLocked()

函数 checkDataOwnerLocked()在文件 ClipboardService.java 中定义，具体实现代码如下所示。

```
    private final void checkDataOwnerLocked(ClipData data, int uid) {
        final int N = data.getItemCount();
```



```
            for (int i=0; i<N; i++) {
                checkItemOwnerLocked(data.getItemAt(i), uid);
            }
        }
private final void checkItemOwnerLocked(ClipData.Item item, int uid) {
        if (item.getUri() != null) {//检查 uri
            checkUriOwnerLocked(item.getUri(), uid);
        }
        Intent intent = item.getIntent();
        //getData()函数返回的也是一个 uri，因此这里实际上检查的也是 uri
        if (intent != null && intent.getData() != null) {
            checkUriOwnerLocked(intent.getData(), uid);
        }
}
```

由此可知，权限检查就是针对 uri 进行的，因为 uri 所指向的数据可能是系统内部使用或私密的。接下来分析文件 ClipboardService.java 中的函数 checkUriOwnerLocked()，具体实现代码如下所示。

```
private final void checkUriOwnerLocked(Uri uri, int uid) {
    if (!"content".equals(uri.getScheme())) {
        return;
    }
    long ident = Binder.clearCallingIdentity();
    try {
        mAm.checkGrantUriPermission(uid, null, uri, Intent.FLAG_GRANT_READ_URI_PERMISSION);
    } catch (RemoteException e) {
    } finally {
        Binder.restoreCallingIdentity(ident);
    }
}
```

通过上述代码，检查 copy 方是否有读取 uri 的权限。

（3）分析函数 clearActiveOwnersLocked()

函数 clearActiveOwnersLocked()在文件 ClipboardService.java 中定义，具体实现代码如下所示。

```
private final void addActiveOwnerLocked(int uid, String pkg) {
    final IPackageManager pm = AppGlobals.getPackageManager();
    final int targetUserHandle = UserHandle.getCallingUserId();
    final long oldIdentity = Binder.clearCallingIdentity();
    try {
        PackageInfo pi = pm.getPackageInfo(pkg, 0, targetUserHandle);
        if (pi == null) {
            throw new IllegalArgumentException("Unknown package " + pkg);
        }
        if (!UserHandle.isSameApp(pi.applicationInfo.uid, uid)) {
            throw new SecurityException("Calling uid " + uid
                    + " does not own package " + pkg);
        }
    } catch (RemoteException e) {
    } finally {
        Binder.restoreCallingIdentity(oldIdentity);
```

```
    }
    PerUserClipboard clipboard = getClipboard();
    if (clipboard.primaryClip != null && !clipboard.activePermissionOwners.contains(pkg)) {
        final int N = clipboard.primaryClip.getItemCount();
        for (int i=0; i<N; i++) {
            grantItemLocked(clipboard.primaryClip.getItemAt(i), pkg);
        }
        clipboard.activePermissionOwners.add(pkg);
    }
}
```

再看文件 ClipboardService.java 中的函数 grantUriLocked()，具体实现代码如下所示。

```
private final void grantUriLocked(Uri uri, String pkg) {
    long ident = Binder.clearCallingIdentity();
    try {
        mAm.grantUriPermissionFromOwner(mPermissionOwner, Process.myUid(), pkg, uri,
                Intent.FLAG_GRANT_READ_URI_PERMISSION);
    } catch (RemoteException e) {
    } finally {
        Binder.restoreCallingIdentity(ident);
    }
}
```

当客户端使用完毕后就需要撤销授权，这个工作是在函数 setPrimaryClip()的 clearActiveOwnersLocked 中完成的。当为剪贴板设置新的 ClipData 时，自然需要将与旧 ClipData 相关的权限撤销。函数 clearActiveOwnersLocked()在文件 ClipboardService.java 中定义，具体实现代码如下所示。

```
    private final void clearActiveOwnersLocked() {
        PerUserClipboard clipboard = getClipboard();
        clipboard.activePermissionOwners.clear();
        if (clipboard.primaryClip == null) {
            return;
        }
        final int N = clipboard.primaryClip.getItemCount();
        for (int i=0; i<N; i++) {
            revokeItemLocked(clipboard.primaryClip.getItemAt(i));
        }
    }
}
```

第 9 章　应用程序进程详解

在 Android 5.0 系统中，在启动应用程序过程中不但可以获得虚拟机实例外，还可以获得一个消息循环和一个 Binder 线程池。这样在应用程序中运行的组件，可以使用系统的信息处理机制和 Binder 通信机制实现自己的业务逻辑。本章将详细分析 Android 5.0 中的应用程序进程的核心知识和具体架构原理。

9.1　创建应用程序

在 Android 系统中，当 ActivityManagerService 创建新进程来启动某个应用程序组件时，会调用类 ActivityManagerService 中的函数 startProcessLocked()向孵化进程 Zygote 发送创建应用程序进程的请求。本节将详细讲解在 Android 5.0 系统中创建应用程序的过程。

9.1.1　发送创建请求

函数 startProcessLocked()在文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中定义，具体实现代码如下所示。

```
private final void startProcessLocked(ProcessRecord app,
            String hostingType, String hostingNameStr) {
        if (app.pid > 0 && app.pid != MY_PID) {
            synchronized (mPidsSelfLocked) {
                mPidsSelfLocked.remove(app.pid);
                mHandler.removeMessages(PROC_START_TIMEOUT_MSG, app);
            }
            app.setPid(0);
        }

        if (DEBUG_PROCESSES && mProcessesOnHold.contains(app)) Slog.v(TAG,
                "startProcessLocked removing on hold: " + app);
        mProcessesOnHold.remove(app);

        updateCpuStats();

        System.arraycopy(mProcDeaths, 0, mProcDeaths, 1, mProcDeaths.length-1);
        mProcDeaths[0] = 0;
        //获取创建应用程序进程的用户 ID 和用户组 ID
        try {
            int uid = app.uid;
```

```
int[] gids = null;
int mountExternal = Zygote.MOUNT_EXTERNAL_NONE;
if (!app.isolated) {
    int[] permGids = null;
    try {
        final PackageManager pm = mContext.getPackageManager();
        permGids = pm.getPackageGids(app.info.packageName);

        if (Environment.isExternalStorageEmulated()) {
            if (pm.checkPermission(
                    android.Manifest.permission.ACCESS_ALL_EXTERNAL_STORAGE,
                    app.info.packageName) == PERMISSION_GRANTED) {
                mountExternal = Zygote.MOUNT_EXTERNAL_MULTIUSER_ALL;
            } else {
                mountExternal = Zygote.MOUNT_EXTERNAL_MULTIUSER;
            }
        }
    } catch (PackageManager.NameNotFoundException e) {
        Slog.w(TAG, "Unable to retrieve gids", e);
    }

    /*
     * Add shared application GID so applications can share some
     * resources like shared libraries
     */
    if (permGids == null) {
        gids = new int[1];
    } else {
        gids = new int[permGids.length + 1];
        System.arraycopy(permGids, 0, gids, 1, permGids.length);
    }
    gids[0] = UserHandle.getSharedAppGid(UserHandle.getAppId(uid));
}
if (mFactoryTest != SystemServer.FACTORY_TEST_OFF) {
    if (mFactoryTest == SystemServer.FACTORY_TEST_LOW_LEVEL
            && mTopComponent != null
            && app.processName.equals(mTopComponent.getPackageName())) {
        uid = 0;
    }
    if (mFactoryTest == SystemServer.FACTORY_TEST_HIGH_LEVEL
            && (app.info.flags&ApplicationInfo.FLAG_FACTORY_TEST) != 0) {
        uid = 0;
    }
}
int debugFlags = 0;
if ((app.info.flags & ApplicationInfo.FLAG_DEBUGGABLE) != 0) {
    debugFlags |= Zygote.DEBUG_ENABLE_DEBUGGER;
    debugFlags |= Zygote.DEBUG_ENABLE_CHECKJNI;
}
```

```
            if ((app.info.flags & ApplicationInfo.FLAG_VM_SAFE_MODE) != 0 ||
                Zygote.systemInSafeMode == true) {
                debugFlags |= Zygote.DEBUG_ENABLE_SAFEMODE;
            }
            if ("1".equals(SystemProperties.get("debug.checkjni"))) {
                debugFlags |= Zygote.DEBUG_ENABLE_CHECKJNI;
            }
            if ("1".equals(SystemProperties.get("debug.jni.logging"))) {
                debugFlags |= Zygote.DEBUG_ENABLE_JNI_LOGGING;
            }
            if ("1".equals(SystemProperties.get("debug.assert"))) {
                debugFlags |= Zygote.DEBUG_ENABLE_ASSERT;
            }
            //调用函数 start()创建应用程序进程
            Process.ProcessStartResult startResult =
Process.start("android.app.ActivityThread",
                    app.processName, uid, uid, gids, debugFlags, mountExternal,
                    app.info.targetSdkVersion, app.info.seinfo, null);
            BatteryStatsImpl bs = app.batteryStats.getBatteryStats();
            synchronized (bs) {
                if (bs.isOnBattery()) {
                    app.batteryStats.incStartsLocked();
                }
            }

            EventLog.writeEvent(EventLogTags.AM_PROC_START,
                    UserHandle.getUserId(uid), startResult.pid, uid,
                    app.processName, hostingType,
                    hostingNameStr != null ? hostingNameStr : "");
            if (app.persistent) {
                Watchdog.getInstance().processStarted(app.processName, startResult.pid);
            }
            StringBuilder buf = mStringBuilder;
            buf.setLength(0);
            buf.append("Start proc ");
            buf.append(app.processName);
            buf.append(" for ");
            buf.append(hostingType);
            if (hostingNameStr != null) {
                buf.append(" ");
                buf.append(hostingNameStr);
            }
            buf.append(": pid=");
            buf.append(startResult.pid);
            buf.append(" uid=");
            buf.append(uid);
            buf.append(" gids={");
            if (gids != null) {
                for (int gi=0; gi<gids.length; gi++) {
```

```
                    if (gi != 0) buf.append(", ");
                    buf.append(gids[gi]);
                }
            }
            buf.append("}");
            Slog.i(TAG, buf.toString());
            app.setPid(startResult.pid);
            app.usingWrapper = startResult.usingWrapper;
            app.removed = false;
            synchronized (mPidsSelfLocked) {
                this.mPidsSelfLocked.put(startResult.pid, app);
                Message msg = mHandler.obtainMessage(PROC_START_TIMEOUT_MSG);
                msg.obj = app;
                mHandler.sendMessageDelayed(msg, startResult.usingWrapper
                        ? PROC_START_TIMEOUT_WITH_WRAPPER : PROC_START_TIMEOUT);
            }
        } catch (RuntimeException e) {
            app.setPid(0);
            Slog.e(TAG, "Failure starting process " + app.processName, e);
        }
    }
```

9.1.2 保存启动参数

类 Process 中的函数 start()在文件 frameworks/base/core/java/android/os/Process.java 中定义，具体实现代码如下所示。

```
public static final ProcessStartResult start(final String processClass,
                              final String niceName,
                              int uid, int gid, int[] gids,
                              int debugFlags, int mountExternal,
                              int targetSdkVersion,
                              String seInfo,
                              String[] zygoteArgs) {
    try {
        //调用函数 startViaZygote()让 Zygote 进程创建一个应用程序进程
        return startViaZygote(processClass, niceName, uid, gid, gids,
                debugFlags, mountExternal, targetSdkVersion, seInfo, zygoteArgs);
    } catch (ZygoteStartFailedEx ex) {
        Log.e(LOG_TAG,
                "Starting VM process through Zygote failed");
         throw new RuntimeException(
                "Starting VM process through Zygote failed", ex);
     }
}
```

在上述代码中用到了函数 startViaZygote()，功能是将要创建的应用程序进程的启动参数保存在字符串列表 argsForZygote 中，并调用函数 zygoteSendArgsAndGetResult()请求进程 Zygote 创建应用程序。函数 startViaZygote()在文件 frameworks/base/core/java/android/os/Process.java 中定义，具体实现代码如

下所示。

```
private static ProcessStartResult startViaZygote(final String processClass,
                                                  final String niceName,
                                                  final int uid, final int gid,
                                                  final int[] gids,
                                                  int debugFlags, int mountExternal,
                                                  int targetSdkVersion,
                                                  String seInfo,
                                                  String[] extraArgs)
                                                  throws ZygoteStartFailedEx {
        synchronized(Process.class) {
            ArrayList<String> argsForZygote = new ArrayList<String>();
            // --runtime-init, --setuid=, --setgid=,
            // and --setgroups= must go first
            argsForZygote.add("--runtime-init");
            argsForZygote.add("--setuid=" + uid);
            argsForZygote.add("--setgid=" + gid);
            if ((debugFlags & Zygote.DEBUG_ENABLE_JNI_LOGGING) != 0) {
                argsForZygote.add("--enable-jni-logging");
            }
            if ((debugFlags & Zygote.DEBUG_ENABLE_SAFEMODE) != 0) {
                argsForZygote.add("--enable-safemode");
            }
            if ((debugFlags & Zygote.DEBUG_ENABLE_DEBUGGER) != 0) {
                argsForZygote.add("--enable-debugger");
            }
            if ((debugFlags & Zygote.DEBUG_ENABLE_CHECKJNI) != 0) {
                argsForZygote.add("--enable-checkjni");
            }
            if ((debugFlags & Zygote.DEBUG_ENABLE_ASSERT) != 0) {
                argsForZygote.add("--enable-assert");
            }
            if (mountExternal == Zygote.MOUNT_EXTERNAL_MULTIUSER) {
                argsForZygote.add("--mount-external-multiuser");
            } else if (mountExternal == Zygote.MOUNT_EXTERNAL_MULTIUSER_ALL) {
                argsForZygote.add("--mount-external-multiuser-all");
            }
            argsForZygote.add("--target-sdk-version=" + targetSdkVersion);
            //argsForZygote.add("--enable-debugger");
            if (gids != null && gids.length > 0) {
                StringBuilder sb = new StringBuilder();
                sb.append("--setgroups=");
                int sz = gids.length;
                for (int i = 0; i < sz; i++) {
                    if (i != 0) {
                        sb.append(',');
                    }
                    sb.append(gids[i]);
                }
```

```
                argsForZygote.add(sb.toString());
            }
            if (niceName != null) {
                argsForZygote.add("--nice-name=" + niceName);
            }
            if (seInfo != null) {
                argsForZygote.add("--seinfo=" + seInfo);
            }
            argsForZygote.add(processClass);
            if (extraArgs != null) {
                for (String arg : extraArgs) {
                    argsForZygote.add(arg);
                }
            }
            //请求进程 Zygote 创建应用程序
            return zygoteSendArgsAndGetResult(argsForZygote);
        }
    }
```

在上述代码中，通过函数 zygoteSendArgsAndGetResult()调用 Zygote 进程创建了一个指定的应用程序。

9.1.3 创建指定的应用程序

函数 zygoteSendArgsAndGetResult()在文件 frameworks/base/core/java/android/os/Process.java 中定义，具体实现代码如下所示。

```
private static ProcessStartResult zygoteSendArgsAndGetResult(ArrayList<String> args)
        throws ZygoteStartFailedEx {
    //调用函数 openZygoteSocketIfNeeded()创建一个连接到 Zygote 进程的本地对象 LocalSocket
    openZygoteSocketIfNeeded();
    try {
        /**
         * 将要创建的应用程序进程启动参数列表写入到本地对象 LocalSocket 中
         * Zygote 进程接收到数据之后会创建一个新的应用程序进程
         *将创建的进程 pid 返回给 ActivityManagerService
         */
        sZygoteWriter.write(Integer.toString(args.size()));
        sZygoteWriter.newLine();
        int sz = args.size();
        for (int i = 0; i < sz; i++) {
            String arg = args.get(i);
            if (arg.indexOf('\n') >= 0) {
                throw new ZygoteStartFailedEx(
                        "embedded newlines not allowed");
            }
            sZygoteWriter.write(arg);
            sZygoteWriter.newLine();
        }
        sZygoteWriter.flush();
```

```
            ProcessStartResult result = new ProcessStartResult();
            result.pid = sZygoteInputStream.readInt();
            if (result.pid < 0) {
                throw new ZygoteStartFailedEx("fork() failed");
            }
            result.usingWrapper = sZygoteInputStream.readBoolean();
            return result;
        } catch (IOException ex) {
            try {
                if (sZygoteSocket != null) {
                    sZygoteSocket.close();
                }
            } catch (IOException ex2) {
                Log.e(LOG_TAG,"I/O exception on routine close", ex2);
            }
            sZygoteSocket = null;
            throw new ZygoteStartFailedEx(ex);
        }
    }
```

在上述代码中用到了函数 openZygoteSocketIfNeeded()，功能是创建一个连接到 Zygote 进程的本地对象 LocalSocket。

9.1.4　创建本地对象 LocalSocket

函数 openZygoteSocketIfNeeded()在文件 frameworks/base/core/java/android/os/Process.java 中定义，具体实现代码如下所示。

```
private static void openZygoteSocketIfNeeded()
            throws ZygoteStartFailedEx {
        int retryCount;
        if (sPreviousZygoteOpenFailed) {
            retryCount = 0;
         } else {
            retryCount = 10;
        }
        for (int retry = 0
                ; (sZygoteSocket == null) && (retry < (retryCount + 1))
                ; retry++ ) {
            if (retry > 0) {
                try {
                    Log.i("Zygote", "Zygote not up yet, sleeping...");
                    Thread.sleep(ZYGOTE_RETRY_MILLIS);
                } catch (InterruptedException ex) {
                }
            }
            try {
                //创建一个保存在 sZygoteSocket 中的 LocalSocket 对象
                sZygoteSocket = new LocalSocket();
```

```
            //将创建的 LocalSocket 对象和名为 ZYGOTE_SOCKET 的 Zygote 进程建立连接
            sZygoteSocket.connect(new LocalSocketAddress(ZYGOTE_SOCKET,
                    LocalSocketAddress.Namespace.RESERVED));
            //将获得的 LocalSocket 对象 sZygoteSocket 的输入流保存在变量 sZygoteInputStream 中
            sZygoteInputStream
                    = new DataInputStream(sZygoteSocket.getInputStream());
            //将获得的 LocalSocket 对象 sZygoteSocket 的输出流保存在变量 sZygoteWriter 中
            sZygoteWriter =
                new BufferedWriter(
                        new OutputStreamWriter(
                                sZygoteSocket.getOutputStream()),
                        256);
            Log.i("Zygote", "Process: zygote socket opened");
            sPreviousZygoteOpenFailed = false;
            break;
        } catch (IOException ex) {
            if (sZygoteSocket != null) {
                try {
                    sZygoteSocket.close();
                } catch (IOException ex2) {
                     Log.e(LOG_TAG,"I/O exception on close after exception",
                            ex2);
                }
            }
            sZygoteSocket = null;
        }
    }
    if (sZygoteSocket == null) {
        sPreviousZygoteOpenFailed = true;
        throw new ZygoteStartFailedEx("connect failed");
    }
}
```

在上述代码中，sZygoteSocket 是一个 LocalSocket 类型的成员变量，能够连接 Zygote 进程中的名为 zygote 的 Socket，这个 Socket 和设备文件/dev/socket/zygote 相对应。

9.1.5 接收创建新应用程序的请求

接下来 Zygote 进程会在函数 runSelectLoop()中接收一个创建新应用程序的要求。函数 runSelectLoop()在文件 frameworks/base/core/java/com/android/internal/os/ZygoteInit.java 中定义，具体实现代码如下所示。

```
private static void runSelectLoop() throws MethodAndArgsCaller {
    ArrayList<FileDescriptor> fds = new ArrayList<FileDescriptor>();
    ArrayList<ZygoteConnection> peers = new ArrayList<ZygoteConnection>();
    FileDescriptor[] fdArray = new FileDescriptor[4];

    fds.add(sServerSocket.getFileDescriptor());
    peers.add(null);
```

```
        int loopCount = GC_LOOP_COUNT;
        while (true) {
            int index;

            if (loopCount <= 0) {
                gc();
                loopCount = GC_LOOP_COUNT;
            } else {
                loopCount--;
            }


            try {
                fdArray = fds.toArray(fdArray);
                index = selectReadable(fdArray);
            } catch (IOException ex) {
                throw new RuntimeException("Error in select()", ex);
            }

            if (index < 0) {
                throw new RuntimeException("Error in select()");
            } else if (index == 0) {
                ZygoteConnection newPeer = acceptCommandPeer();
                peers.add(newPeer);
                fds.add(newPeer.getFileDesciptor());
            } else {
                boolean done;
                done = peers.get(index).runOnce();
                if (done) {
                    peers.remove(index);
                    fds.remove(index);
                }
            }
        }
    }
```

在上述代码中，会调用函数 runOnce()处理接收到创建新应用程序的要求。函数 runOnce()在文件 frameworks/base/core/java/com/android/internal/os/ZygoteConnection.java 中定义，具体实现代码如下所示。

```
boolean runOnce() throws ZygoteInit.MethodAndArgsCaller {

    String args[];
    Arguments parsedArgs = null;
    FileDescriptor[] descriptors;

    try {
        args = readArgumentList();//获得启动要创建应用程序进程的参数
        descriptors = mSocket.getAncillaryFileDescriptors();
    } catch (IOException ex) {
        Log.w(TAG, "IOException on command socket " + ex.getMessage());
        closeSocket();
```

```
        return true;
    }

    if (args == null) {
        // EOF reached
        closeSocket();
        return true;
    }

    /** the stderr of the most recent request, if avail */
    PrintStream newStderr = null;

    if (descriptors != null && descriptors.length >= 3) {
        newStderr = new PrintStream(
                new FileOutputStream(descriptors[2]));
    }

    int pid = -1;
    FileDescriptor childPipeFd = null;
    FileDescriptor serverPipeFd = null;

    try {
        parsedArgs = new Arguments(args);

        applyUidSecurityPolicy(parsedArgs, peer, peerSecurityContext);
        applyRlimitSecurityPolicy(parsedArgs, peer, peerSecurityContext);
        applyCapabilitiesSecurityPolicy(parsedArgs, peer, peerSecurityContext);
        applyInvokeWithSecurityPolicy(parsedArgs, peer, peerSecurityContext);
        applyseInfoSecurityPolicy(parsedArgs, peer, peerSecurityContext);

        applyDebuggerSystemProperty(parsedArgs);
        applyInvokeWithSystemProperty(parsedArgs);

        int[][] rlimits = null;

        if (parsedArgs.rlimits != null) {
            rlimits = parsedArgs.rlimits.toArray(intArray2d);
        }

        if (parsedArgs.runtimeInit && parsedArgs.invokeWith != null) {
            FileDescriptor[] pipeFds = Libcore.os.pipe();
            childPipeFd = pipeFds[1];
            serverPipeFd = pipeFds[0];
            ZygoteInit.setCloseOnExec(serverPipeFd, true);
        }
        //调用函数 forkAndSpecialize()创建应用程序进程
        pid = Zygote.forkAndSpecialize(parsedArgs.uid, parsedArgs.gid, parsedArgs.gids,
                parsedArgs.debugFlags, rlimits, parsedArgs.mountExternal, parsedArgs.seInfo,
                parsedArgs.niceName);
    } catch (IOException ex) {
        logAndPrintError(newStderr, "Exception creating pipe", ex);
```

```
    } catch (ErrnoException ex) {
        logAndPrintError(newStderr, "Exception creating pipe", ex);
    } catch (IllegalArgumentException ex) {
        logAndPrintError(newStderr, "Invalid zygote arguments", ex);
    } catch (ZygoteSecurityException ex) {
        logAndPrintError(newStderr,
                "Zygote security policy prevents request: ", ex);
    }

    try {
        if (pid == 0) {
            IoUtils.closeQuietly(serverPipeFd);
            serverPipeFd = null;
            handleChildProc(parsedArgs, descriptors, childPipeFd, newStderr);

            return true;
        } else {
            IoUtils.closeQuietly(childPipeFd);
            childPipeFd = null;
            return handleParentProc(pid, descriptors, serverPipeFd, parsedArgs);
        }
    } finally {
        IoUtils.closeQuietly(childPipeFd);
        IoUtils.closeQuietly(serverPipeFd);
    }
}
```

在上述代码中，通过函数 readArgumentList()获得启动要创建应用程序进程的参数，并通过函数 forkAndSpecialize()创建了这个要启动应用程序的进程。其中，函数 readArgumentList()在文件 frameworks/base/core/java/com/android/internal/os/ZygoteConnection.java 中定义，具体实现代码如下所示。

```
private String[] readArgumentList()
        throws IOException {

    /**
     * See android.os.Process.zygoteSendArgsAndGetPid()
     * Presently the wire format to the zygote process is:
     * a) a count of arguments (argc, in essence)
     * b) a number of newline-separated argument strings equal to count
     *
     * After the zygote process reads these it will write the pid of
     * the child or -1 on failure
     */

    int argc;

    try {
        String s = mSocketReader.readLine();

        if (s == null) {
```

```
                return null;
            }
            argc = Integer.parseInt(s);
        } catch (NumberFormatException ex) {
            Log.e(TAG, "invalid Zygote wire format: non-int at argc");
            throw new IOException("invalid wire format");
         }

        if (argc > MAX_ZYGOTE_ARGC) {
            throw new IOException("max arg count exceeded");
        }

        String[] result = new String[argc];
        for (int i = 0; i < argc; i++) {
            result[i] = mSocketReader.readLine();
            if (result[i] == null) {
                throw new IOException("truncated request");
            }
         }

        return result;
}
```

函数 forkAndSpecialize()在文件 libcore/dalvik/src/main/java/dalvik/system/Zygote.java 中定义，具体实现代码如下所示。

```
public static int forkAndSpecialize(int uid, int gid, int[] gids, int debugFlags,
            int[][] rlimits, int mountExternal, String seInfo, String niceName) {
        preFork();
        int pid = nativeForkAndSpecialize(
                uid, gid, gids, debugFlags, rlimits, mountExternal, seInfo, niceName);
        postFork();
        return pid;
}
```

在上述代码中，当创建一个进程的子进程时，如果返回值为 0，则表示在新创建的进程中执行。此时需要调用函数 handleChildProc()来启动这个子进程，并在 handleChildProc 中调用函数 zygoteInit()在新创建的应用程序进程中初始化运行库，这样便可以启动一个 Binder 线程池。

9.2　启动线程池

在创建新应用程序完毕之前，需要调用类 RuntimeInit 中的函数 nativeZygoteInit()启动一个新的 Binder 线程池，具体启动流程如下所示。

（1）调用类 RuntimeInit 中的函数 nativeZygoteInit()，此函数在文件 frameworks/base/core/java/com/android/internal/os/RuntimeInit.java 中定义，对应的实现代码如下所示。

```
public class RuntimeInit {
    private final static String TAG = "AndroidRuntime";
```

```
    private final static boolean DEBUG = false;

    /** true if commonInit() has been called */
    private static boolean initialized;

    private static IBinder mApplicationObject;

    private static volatile boolean mCrashing = false;

    private static final native void nativeZygoteInit();
    private static final native void nativeFinishInit();
```

（2）函数 nativeZygoteInit()是一个 JNI 函数，在文件 frameworks/base/core/jni/AndroidRuntime.cpp 中定义，对应代码如下所示。

```
static void com_android_internal_os_RuntimeInit_nativeZygoteInit(JNIEnv* env, jobject clazz)
{
    gCurRuntime->onZygoteInit();
}
```

（3）在上述实现代码中，gCurRuntime 是一个全局变量，上述代码用到了 gCurRuntime 的成员函数 onZygoteInit()启动了一个 Binder 线程池。函数 onZygoteInit()在文件 frameworks/base/cmds/app_process/app_main.cpp 中定义，具体实现代码如下所示。

```
virtual void onZygoteInit()
{
    atrace_set_tracing_enabled(true);

    sp<ProcessState> proc = ProcessState::self();
    ALOGV("App process: starting thread pool.\n");
    //调用函数 startThreadPool()启动一个 Binder 线程池
    proc->startThreadPool();
}
```

（4）在上述代码中，当调用函数 startThreadPool()启动一个 Binder 线程池后，当前应用程序进程就可以通过 Binder 机制和其他进程实现通信。函数 startThreadPool()在文件 frameworks/native/libs/binder/ProcessState.cpp 中定义，具体实现代码如下所示。

```
void ProcessState::startThreadPool()
{
    AutoMutex _l(mLock);
    if (!mThreadPoolStarted) {
        mThreadPoolStarted = true;//默认值为 false
        spawnPooledThread(true);
    }
}
```

（5）在上述代码中，mThreadPoolStarted 的默认值为 false。当第一次调用函数 startThreadPool()时，会在当前进程中启动 Binder 线程池，并将 mThreadPoolStarted 设置为 true，这样做的目的是防止以后重复启动 Binder 线程池。

9.3 创建信息循环

当创建新应用程序进程完毕以后，会调用函数 invokeStaticMain()将类 ActivityThread 的函数 main()设置为新程序的入口函数。当使用函数 main()时，会在当前程序的进程中建立一个信息循环。

接下来首先看函数 invokeStaticMain()的具体实现，此函数在文件 frameworks/base/core/java/com/android/internal/os/RuntimeInit.java 中定义，具体实现代码如下所示。

```
private static void invokeStaticMain(String className, String[] argv)
        throws ZygoteInit.MethodAndArgsCaller {
    Class<?> cl;

     try {
        cl = Class.forName(className);
    } catch (ClassNotFoundException ex) {
        throw new RuntimeException(
                "Missing class when invoking static main " + className,
                ex);
    }

     Method m;
     try {
        //获得静态成员函数 main()，并保存在 Method 对象中
        m = cl.getMethod("main", new Class[] { String[].class });
    } catch (NoSuchMethodException ex) {
        throw new RuntimeException(
                "Missing static main on " + className, ex);
    } catch (SecurityException ex) {
         throw new RuntimeException(
                 "Problem getting static main on " + className, ex);
    }

     int modifiers = m.getModifiers();
    if (! (Modifier.isStatic(modifiers) && Modifier.isPublic(modifiers))) {
        throw new RuntimeException(
                "Main method is not public and static on " + className);
    }
    /*
     *将 method 对象封装在静态成员函数 main()中，并保存在一个 Method 对象中
     * 将 MethodAndArgsCaller 对象作为异常抛给当前程序进程来处理
     */
     throw new ZygoteInit.MethodAndArgsCaller(m, argv);
}
```

静态成员函数 main()在文件 frameworks/base/core/java/com/android/internal/os/RuntimeInit.java 中定义，具体实现代码如下所示。

```
public static final void main(String[] argv) {
    if (argv.length == 2 && argv[1].equals("application")) {
```



```
            if (DEBUG) Slog.d(TAG, "RuntimeInit: Starting application");
            redirectLogStreams();
        } else {
            if (DEBUG) Slog.d(TAG, "RuntimeInit: Starting tool");
        }
         commonInit();
        /*
         * Now that we're running in interpreted code, call back into native code
         * to run the system
         */
        nativeFinishInit();
         if (DEBUG) Slog.d(TAG, "Leaving RuntimeInit!");
}
```

在上述代码中，当函数 main()捕获到 MethodAndArgsCaller 异常后，会调用 MethodAndArgsCaller 成员函数 run()进行后面的处理。接下来看函数 MethodAndArgsCaller()和 run()，这两个函数都是在文件 frameworks/base/core/java/com/android/internal/os/ZygoteInit.java 中定义，具体实现代码如下所示。

```
public static class MethodAndArgsCaller extends Exception
            implements Runnable {
        /** method to call */
        private final Method mMethod;

        /** argument array */
        private final String[] mArgs;

        public MethodAndArgsCaller(Method method, String[] args) {
            mMethod = method;
            mArgs = args;
        }

        public void run() {
            try {
                //执行函数 invoke()，这样就执行了类 android.app.ActivityThread 中的函数 main()
                mMethod.invoke(null, new Object[] { mArgs });
            } catch (IllegalAccessException ex) {
                throw new RuntimeException(ex);
            } catch (InvocationTargetException ex) {
                Throwable cause = ex.getCause();
                if (cause instanceof RuntimeException) {
                    throw (RuntimeException) cause;
                } else if (cause instanceof Error) {
                     throw (Error) cause;
                }
                throw new RuntimeException(ex);
            }
        }
    }
}
```

在上述代码中，变量 mMethod 和 mArgs 是在构造异常对象时传递进来的，其中变量 mMethod 和类 android.app.ActivityThread 中的函数 main()相对应。

第 10 章　ART 机制架构详解

从 Android 5.0 版本开始，Android 应用程序默认的运行环境为 ART。ART 机制与 Dalvik 不同，在 Dalvik 下，应用每次运行时，字节码都需要通过即时编译器转换为机器码，这会拖慢应用的运行效率，而在 ART 环境中，应用在第一次安装时，字节码就会预先编译成机器码，使其成为真正的本地应用。这个过程叫做预编译（Ahead-Of-Time，AOT）。这样，应用的启动（首次）和执行都会变得更加快速。本章将简要介绍 Android ART 机制的基本架构知识。

10.1　分析 ART 的启动过程

传统的 Dalvik 虚拟机其实是一个 Java 虚拟机，只不过它执行的不是 CLASS 文件，而是 DEX 文件。因此，ART 运行时最理想的方式也是实现为一个 Java 虚拟机的形式，这样就可以很容易地将 Dalvik 虚拟机替换掉。ART 运行时就是真的和 Dalvik 虚拟机一样，实现了一套完全兼容 Java 虚拟机的接口。为了方便描述，接下来就将 ART 运行时称为 ART 虚拟机，它和 Dalvik 虚拟机、Java 虚拟机的关系如图 10-1 所示。

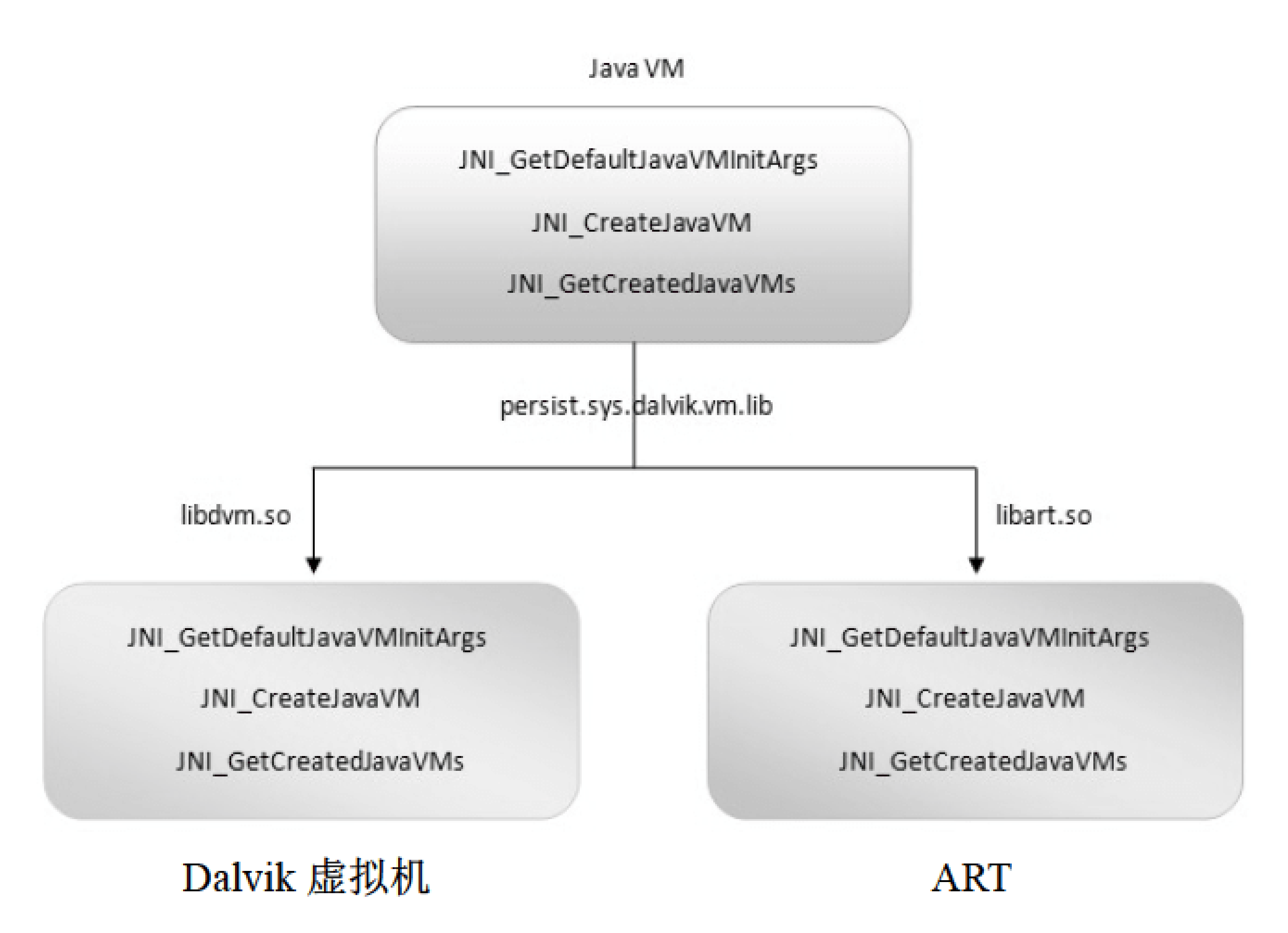


图 10-1　ART、Dalvik 和 Java 虚拟机的关系

从图 10-1 可知，Dalvik 虚拟机和 ART 虚拟机都实现了如下 3 个用来抽象 Java 虚拟机的接口。

（1）JNI_GetDefaultJavaVMInitArgs：获取虚拟机的默认初始化参数。

（2）JNI_CreateJavaVM：在进程中创建虚拟机实例。

（3）JNI_GetCreatedJavaVMs：获取进程中创建的虚拟机实例。

在 Android 系统中，Dalvik 虚拟机实现在 libdvm.so 文件中，ART 虚拟机实现在 libart.so 文件中。

也就是说，文件 libdvm.so 和文件 libart.so 导出了以上 3 个接口供外界调用。

此外，Android 系统还提供了一个系统属性 persist.sys.dalvik.vm.lib，其值等于 libdvm.so 或 libart.so，具体说明如下所示。

☑ 当等于 libdvm.so 时，表示当前用的是 Dalvik 虚拟机。

☑ 当等于 libart.so 时，表示当前用的是 ART 虚拟机。

上面介绍了 Dalvik 虚拟机和 ART 虚拟机的共同之处，下面介绍两者的不同之处。Dalvik 虚拟机执行的是 DEX 字节码，ART 虚拟机执行的是本地机器码。这意味着 Dalvik 虚拟机包含有一个解释器，用来执行 DEX 字节码。当然，Android 从 2.2 开始，也包含有 JIT（Just-In-Time），用来在运行时动态地将执行频率很高的 DEX 字节码翻译成本地机器码，然后再执行。通过 JIT，就可以有效地提高 Dalvik 虚拟机的执行效率。但是，将 DEX 字节码翻译成本地机器码是发生在应用程序的运行过程中的，并且应用程序每一次重新运行时，都要重做这个翻译工作。因此，即使采用了 JIT，Dalvik 虚拟机的总体性能还是不能与直接执行本地机器码的 ART 虚拟机相比。

10.1.1　运行 app_process 进程

启动过程从 init.rc 文件开始，在文件 init.rc 中由如下代码表示启动 Zygote。

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote--start-system-server
```

init 进程根据此指令执行 app_process（frameworks/base/cmds/app_process/app_main.cpp），也就是 Zygote 了。当 Android 系统启动时会创建一个 Zygote 进程，作为应用程序的进程孵化器，并且在启动 Zygote 进程的过程中会创建一个 Dalvik 虚拟机。Zygote 进程是通过复制自己来创建新的应用程序进程的，这意味着 Zygote 进程会将自己的 Dalvik 虚拟机复制给应用程序进程。上述方式可以大大提高应用程序的启动速度，因为这种方式避免了每一个应用程序进程在启动时都要去创建一个 Dalvik。事实上，Zygote 进程通过自我复制的方式来创建应用程序进程，省去的不仅仅是应用程序进程创建 Dalvik 虚拟机的时间，还能省去应用程序进程加载各种系统库和系统资源的时间，因为它们在 Zygote 进程中已经加载过了，并且也会连同 Dalvik 虚拟机一起复制到应用程序进程中去。这也是 ART 优于 Dalvik 的原因。

当 Android 系统启动 init 进程时会运行 app_process 进程，在文件/frameworks/base/cmds/app_process/app_main.cpp 中定义了 app_process 进程的具体实现，在主函数 main()中会启动 Zygote，对应代码为如下加粗部分。

```
if (niceName && *niceName) {
    setArgv0(argv0, niceName);
    set_process_name(niceName);
}

runtime.mParentDir = parentDir;

if (zygote) {
    runtime.start("com.android.internal.os.ZygoteInit",
            startSystemServer ? "start-system-server" : "");
} else if (className) {
    runtime.mClassName = className;
    runtime.mArgC = argc - i;
```

```
        runtime.mArgV = argv + i;
        runtime.start("com.android.internal.os.RuntimeInit",
                application ? "application" : "tool");
    } else {
        fprintf(stderr, "Error: no class name or --zygote supplied.\n");
        app_usage();
        LOG_ALWAYS_FATAL("app_process: no class name or --zygote supplied.");
        return 10;
    }
}
```

在上述代码中，runtime 是 AppRuntime 的实例，AppRuntime 继承自 AndroidRuntime。类 AndroidRuntime 中的函数 start()在文件 frameworks/base/core/jni/AndroidRuntime.cpp 中定义，具体实现代码如下所示。

```
void AndroidRuntime::start(const char* className, const char* options)
{
    ALOGD("\n>>>>>> AndroidRuntime START %s <<<<<<\n",
            className != NULL ? className : "(unknown)");

    /*
     * 'startSystemServer == true' means runtime is obsolete and not run from
     * init.rc anymore, so we print out the boot start event here.
     */
    if (strcmp(options, "start-system-server") == 0) {
        /* track our progress through the boot sequence */
        const int LOG_BOOT_PROGRESS_START = 3000;
        LOG_EVENT_LONG(LOG_BOOT_PROGRESS_START,
                       ns2ms(systemTime(SYSTEM_TIME_MONOTONIC)));
    }

    const char* rootDir = getenv("ANDROID_ROOT");
    if (rootDir == NULL) {
        rootDir = "/system";
        if (!hasDir("/system")) {
            LOG_FATAL("No root directory specified, and /android does not exist.");
            return;
        }
        setenv("ANDROID_ROOT", rootDir, 1);
    }

    //const char* kernelHack = getenv("LD_ASSUME_KERNEL");
    //ALOGD("Found LD_ASSUME_KERNEL='%s'\n", kernelHack);

    /* start the virtual machine */
    JniInvocation jni_invocation;
    jni_invocation.Init(NULL);
    JNIEnv* env;
    if (startVm(&mJavaVM, &env) != 0) {
        return;
    }
    onVmCreated(env);
```

```
    /*
     * Register android functions
     */
    if (startReg(env) < 0) {
        ALOGE("Unable to register all android natives\n");
        return;
    }

    /*
     * We want to call main() with a String array with arguments in it.
     * At present we have two arguments, the class name and an option string.
     * Create an array to hold them
     */
    jclass stringClass;
    jobjectArray strArray;
    jstring classNameStr;
    jstring optionsStr;

    stringClass = env->FindClass("java/lang/String");
    assert(stringClass != NULL);
    strArray = env->NewObjectArray(2, stringClass, NULL);
    assert(strArray != NULL);
    classNameStr = env->NewStringUTF(className);
    assert(classNameStr != NULL);
    env->SetObjectArrayElement(strArray, 0, classNameStr);
    optionsStr = env->NewStringUTF(options);
    env->SetObjectArrayElement(strArray, 1, optionsStr);

    /*
     * Start VM.  This thread becomes the main thread of the VM, and will
     * not return until the VM exits
     */
    char* slashClassName = toSlashClassName(className);
    jclass startClass = env->FindClass(slashClassName);
    if (startClass == NULL) {
        ALOGE("JavaVM unable to locate class '%s'\n", slashClassName);
        /* keep going */
    } else {
        jmethodID startMeth = env->GetStaticMethodID(startClass, "main",
            "([Ljava/lang/String;)V");
        if (startMeth == NULL) {
            ALOGE("JavaVM unable to find main() in '%s'\n", className);
            /* keep going */
        } else {
            env->CallStaticVoidMethod(startClass, startMeth, strArray);

#if 0
            if (env->ExceptionCheck())
                threadExitUncaughtException(env);
#endif
```

```
            }
        }
        free(slashClassName);

        ALOGD("Shutting down VM\n");
        if (mJavaVM->DetachCurrentThread() != JNI_OK)
            ALOGW("Warning: unable to detach main thread\n");
        if (mJavaVM->DestroyJavaVM() != 0)
            ALOGW("Warning: VM did not shut down cleanly\n");
}
```

在上述代码中，JniInvocation jni_invocation;用于声明类 JniInvocation 的变量，jni_invocation.Init(NULL);用于调用类 JniInvocation 中的函数 Init()。由此可见，类 AndroidRutime 的成员函数 start()最主要实现了如下 3 个功能。

- ☑ 创建一个 JniInvocation 实例，并且调用其成员函数 Init()来初始化 JNI 环境。
- ☑ 调用 AndroidRutime 类的成员函数 startVm()来创建一个虚拟机及其对应的 JNI 接口，即创建一个 JavaVM 接口和一个 JNIEnv 接口。
- ☑ 通过上述 JavaVM 接口和 JNIEnv 接口在 Zygote 进程中加载指定的 class。

其中，上述前两个功能是最关键的。因此，接下来继续分析它们所对应的函数的实现。

类 JniInvocation 在文件/libnativehelper/JniInvocation.cpp 中定义，函数 Init()的具体实现代码如下所示。

```
bool JniInvocation::Init(const char* library) {
#ifdef HAVE_ANDROID_OS
  char default_library[PROPERTY_VALUE_MAX];
  property_get("persist.sys.dalvik.vm.lib", default_library, "libdvm.so");
#else
  const char* default_library = "libdvm.so";
#endif
  if (library == NULL) {
    library = default_library;
  }

  handle_ = dlopen(library, RTLD_NOW);
  if (handle_ == NULL) {
    ALOGE("Failed to dlopen %s: %s", library, dlerror());
    return false;
  }
  if (!FindSymbol(reinterpret_cast<void**>(&JNI_GetDefaultJavaVMInitArgs_),
                  "JNI_GetDefaultJavaVMInitArgs")) {
    return false;
  }
  if (!FindSymbol(reinterpret_cast<void**>(&JNI_CreateJavaVM_),
                  "JNI_CreateJavaVM")) {
    return false;
  }
  if (!FindSymbol(reinterpret_cast<void**>(&JNI_GetCreatedJavaVMs_),
                  "JNI_GetCreatedJavaVMs")) {
    return false;
  }
```

```
    return true;
}
```

在上述代码中，函数 Init()首先读取系统属性 persist.sys.dalvik.vm.lib 的值。因为系统属性 persist.sys.dalvik.vm.lib 的值等于 libdvm.so 或 libart.so，所以接下来通过函数 dlopen()加载到进程中的是 libdvm.so 或 libart.so。无论加载的是哪一个，都要求导出 JNI_GetDefaultJavaVMInitArgs、JNI_CreateJavaVM 和 JNI_GetCreatedJavaVMs 这 3 个接口，并且分别保存在 JniInvocation 类的 3 个成员变量 JNI_GetDefaultJavaVMInitArgs_、JNI_CreateJavaVM_和 JNI_GetCreatedJavaVMs_中，这 3 个接口也就是前面提到的用来抽象 Java 虚拟机的 3 个接口。

10.1.2　准备启动

回到函数 AndroidRuntime::start()，if (startVm(&mJavaVM, &env) != 0) {用于调用函数 startVm()启动虚拟机。也就是说，类 JniInvocation 的成员函数 Init()实际上就是根据系统属性 persist.sys.dalvik.vm.lib 来初始化 Dalvik 虚拟机或者 ART 虚拟机环境。AndroidRutime 类的成员函数 AndroidRuntime::startVm()的具体实现代码如下所示。

```
int AndroidRuntime::startVm(JavaVM** pJavaVM, JNIEnv** pEnv)
{
    int result = -1;
    JavaVMInitArgs initArgs;
    JavaVMOption opt;
    char propBuf[PROPERTY_VALUE_MAX];
    char stackTraceFileBuf[PROPERTY_VALUE_MAX];
    char dexoptFlagsBuf[PROPERTY_VALUE_MAX];
    char enableAssertBuf[sizeof("-ea:")-1 + PROPERTY_VALUE_MAX];
    char jniOptsBuf[sizeof("-Xjniopts:")-1 + PROPERTY_VALUE_MAX];
    char heapstartsizeOptsBuf[sizeof("-Xms")-1 + PROPERTY_VALUE_MAX];
    char heapsizeOptsBuf[sizeof("-Xmx")-1 + PROPERTY_VALUE_MAX];
    char heapgrowthlimitOptsBuf[sizeof("-XX:HeapGrowthLimit=")-1 + PROPERTY_VALUE_MAX];
    char heapminfreeOptsBuf[sizeof("-XX:HeapMinFree=")-1 + PROPERTY_VALUE_MAX];
    char heapmaxfreeOptsBuf[sizeof("-XX:HeapMaxFree=")-1 + PROPERTY_VALUE_MAX];
    char heaptargetutilizationOptsBuf[sizeof("-XX:HeapTargetUtilization=")-1 + PROPERTY_VALUE_MAX];
    char jitcodecachesizeOptsBuf[sizeof("-Xjitcodecachesize:")-1 + PROPERTY_VALUE_MAX];
    char extraOptsBuf[PROPERTY_VALUE_MAX];
    char* stackTraceFile = NULL;
    bool checkJni = false;
    bool checkDexSum = false;
    bool logStdio = false;
    enum {
      kEMDefault,
      kEMIntPortable,
      kEMIntFast,
      kEMJitCompiler,
    } executionMode = kEMDefault;


    property_get("dalvik.vm.checkjni", propBuf, "");
    if (strcmp(propBuf, "true") == 0) {
```

```
        checkJni = true;
    } else if (strcmp(propBuf, "false") != 0) {
        /* property is neither true nor false; fall back on kernel parameter */
        property_get("ro.kernel.android.checkjni", propBuf, "");
        if (propBuf[0] == '1') {
            checkJni = true;
        }
    }

    property_get("dalvik.vm.execution-mode", propBuf, "");
    if (strcmp(propBuf, "int:portable") == 0) {
        executionMode = kEMIntPortable;
    } else if (strcmp(propBuf, "int:fast") == 0) {
        executionMode = kEMIntFast;
    } else if (strcmp(propBuf, "int:jit") == 0) {
        executionMode = kEMJitCompiler;
    }

    property_get("dalvik.vm.stack-trace-file", stackTraceFileBuf, "");

    property_get("dalvik.vm.check-dex-sum", propBuf, "");
    if (strcmp(propBuf, "true") == 0) {
        checkDexSum = true;
    }

    property_get("log.redirect-stdio", propBuf, "");
    if (strcmp(propBuf, "true") == 0) {
        logStdio = true;
    }

    strcpy(enableAssertBuf, "-ea:");
    property_get("dalvik.vm.enableassertions", enableAssertBuf+4, "");

    strcpy(jniOptsBuf, "-Xjniopts:");
    property_get("dalvik.vm.jniopts", jniOptsBuf+10, "");

    /*exit()线程处理*/
    opt.extraInfo = (void*) runtime_exit;
    opt.optionString = "exit";
    mOptions.add(opt);

    /*fprintf()线程处理*/
    opt.extraInfo = (void*) runtime_vfprintf;
    opt.optionString = "vfprintf";
    mOptions.add(opt);

    /*注册敏感线程框架*/
    opt.extraInfo = (void*) runtime_isSensitiveThread;
    opt.optionString = "sensitiveThread";
    mOptions.add(opt);
```

```
opt.extraInfo = NULL;

/* enable verbose; standard options are { jni, gc, class } */
//options[curOpt++].optionString = "-verbose:jni";
opt.optionString = "-verbose:gc";
mOptions.add(opt);
//options[curOpt++].optionString = "-verbose:class";

/*
 *默认的启动和堆的最大尺寸
 */
strcpy(heapstartsizeOptsBuf, "-Xms");
property_get("dalvik.vm.heapstartsize", heapstartsizeOptsBuf+4, "4m");
opt.optionString = heapstartsizeOptsBuf;
mOptions.add(opt);
strcpy(heapsizeOptsBuf, "-Xmx");
property_get("dalvik.vm.heapsize", heapsizeOptsBuf+4, "16m");
opt.optionString = heapsizeOptsBuf;
mOptions.add(opt);

//增加错误主线程的解释器的堆栈大小：6315322
opt.optionString = "-XX:mainThreadStackSize=24K";
mOptions.add(opt);

//设置最大 JIT 代码缓存大小。0 表示将禁用 JIT
strcpy(jitcodecachesizeOptsBuf, "-Xjitcodecachesize:");
property_get("dalvik.vm.jit.codecachesize", jitcodecachesizeOptsBuf+19,   NULL);
if (jitcodecachesizeOptsBuf[19] != '\0') {
   opt.optionString = jitcodecachesizeOptsBuf;
   mOptions.add(opt);
}

strcpy(heapgrowthlimitOptsBuf, "-XX:HeapGrowthLimit=");
property_get("dalvik.vm.heapgrowthlimit", heapgrowthlimitOptsBuf+20, "");
if (heapgrowthlimitOptsBuf[20] != '\0') {
      opt.optionString = heapgrowthlimitOptsBuf;
      mOptions.add(opt);
}

strcpy(heapminfreeOptsBuf, "-XX:HeapMinFree=");
property_get("dalvik.vm.heapminfree", heapminfreeOptsBuf+16, "");
if (heapminfreeOptsBuf[16] != '\0') {
      opt.optionString = heapminfreeOptsBuf;
      mOptions.add(opt);
}

strcpy(heapmaxfreeOptsBuf, "-XX:HeapMaxFree=");
property_get("dalvik.vm.heapmaxfree", heapmaxfreeOptsBuf+16, "");
if (heapmaxfreeOptsBuf[16] != '\0') {
      opt.optionString = heapmaxfreeOptsBuf;
      mOptions.add(opt);
```

```
    }

    strcpy(heaptargetutilizationOptsBuf, "-XX:HeapTargetUtilization=");
    property_get("dalvik.vm.heaptargetutilization", heaptargetutilizationOptsBuf+26, "");
    if (heaptargetutilizationOptsBuf[26] != '\0') {
        opt.optionString = heaptargetutilizationOptsBuf;
        mOptions.add(opt);
    }

    property_get("ro.config.low_ram", propBuf, "");
    if (strcmp(propBuf, "true") == 0) {
      opt.optionString = "-XX:LowMemoryMode";
      mOptions.add(opt);
    }

    /*
     *启用或禁用 dexopt 特征，如字节码校验和为精确计算 GC 寄存器映射
     */
    property_get("dalvik.vm.dexopt-flags", dexoptFlagsBuf, "");
    if (dexoptFlagsBuf[0] != '\0') {
        const char* opc;
        const char* val;

        opc = strstr(dexoptFlagsBuf, "v=");     /* verification */
        if (opc != NULL) {
            switch (*(opc+2)) {
            case 'n': val = "-Xverify:none"; break;
            case 'r': val = "-Xverify:remote"; break;
            case 'a': val = "-Xverify:all"; break;
            default: val = NULL; break;
            }

            if (val != NULL) {
                opt.optionString = val;
                mOptions.add(opt);
            }
        }

        opc = strstr(dexoptFlagsBuf, "o=");     /* optimization */
        if (opc != NULL) {
            switch (*(opc+2)) {
            case 'n': val = "-Xdexopt:none"; break;
            case 'v': val = "-Xdexopt:verified"; break;
            case 'a': val = "-Xdexopt:all"; break;
            case 'f': val = "-Xdexopt:full"; break;
            default: val = NULL; break;
            }

            if (val != NULL) {
                opt.optionString = val;
                mOptions.add(opt);
```

```
            }
        }

        opc = strstr(dexoptFlagsBuf, "m=y");        /* register map */
        if (opc != NULL) {
            opt.optionString = "-Xgenregmap";
            mOptions.add(opt);

            /* turn on precise GC while we're at it */
            opt.optionString = "-Xgc:precise";
            mOptions.add(opt);
        }
    }

    /*启用调试；设置暂停= Y，暂停 VM 初始化*/
    /* use android ADB transport */
    opt.optionString =
        "-agentlib:jdwp=transport=dt_android_adb,suspend=n,server=y";
    mOptions.add(opt);

    ALOGD("CheckJNI is %s\n", checkJni ? "ON" : "OFF");
    if (checkJni) {
        /*扩展的 JNI 检查*/
        opt.optionString = "-Xcheck:jni";
        mOptions.add(opt);

        /*设置 JNI 全局引用*/
        opt.optionString = "-Xjnigreflimit:2000";
        mOptions.add(opt);

        /* with -Xcheck:jni, this provides a JNI function call trace */
        //opt.optionString = "-verbose:jni";
        //mOptions.add(opt);
    }

    char lockProfThresholdBuf[sizeof("-Xlockprofthreshold:") + sizeof(propBuf)];
    property_get("dalvik.vm.lockprof.threshold", propBuf, "");
    if (strlen(propBuf) > 0) {
      strcpy(lockProfThresholdBuf, "-Xlockprofthreshold:");
      strcat(lockProfThresholdBuf, propBuf);
      opt.optionString = lockProfThresholdBuf;
      mOptions.add(opt);
    }

    /* Force interpreter-only mode for selected opcodes. Eg "1-0a,3c,f1-ff" */
    char jitOpBuf[sizeof("-Xjitop:") + PROPERTY_VALUE_MAX];
    property_get("dalvik.vm.jit.op", propBuf, "");
    if (strlen(propBuf) > 0) {
        strcpy(jitOpBuf, "-Xjitop:");
        strcat(jitOpBuf, propBuf);
        opt.optionString = jitOpBuf;
```

```
        mOptions.add(opt);
    }

    /* Force interpreter-only mode for selected methods */
    char jitMethodBuf[sizeof("-Xjitmethod:") + PROPERTY_VALUE_MAX];
    property_get("dalvik.vm.jit.method", propBuf, "");
    if (strlen(propBuf) > 0) {
        strcpy(jitMethodBuf, "-Xjitmethod:");
        strcat(jitMethodBuf, propBuf);
        opt.optionString = jitMethodBuf;
        mOptions.add(opt);
    }

    if (executionMode == kEMIntPortable) {
        opt.optionString = "-Xint:portable";
        mOptions.add(opt);
    } else if (executionMode == kEMIntFast) {
        opt.optionString = "-Xint:fast";
        mOptions.add(opt);
    } else if (executionMode == kEMJitCompiler) {
        opt.optionString = "-Xint:jit";
        mOptions.add(opt);
    }

    if (checkDexSum) {
        /* perform additional DEX checksum tests */
        opt.optionString = "-Xcheckdexsum";
        mOptions.add(opt);
    }

    if (logStdio) {
        /* convert stdout/stderr to log messages */
        opt.optionString = "-Xlog-stdio";
        mOptions.add(opt);
    }

    if (enableAssertBuf[4] != '\0') {
        /* accept "all" to mean "all classes and packages" */
        if (strcmp(enableAssertBuf+4, "all") == 0)
            enableAssertBuf[3] = '\0';
        ALOGI("Assertions enabled: '%s'\n", enableAssertBuf);
        opt.optionString = enableAssertBuf;
        mOptions.add(opt);
    } else {
        ALOGV("Assertions disabled\n");
    }

    if (jniOptsBuf[10] != '\0') {
        ALOGI("JNI options: '%s'\n", jniOptsBuf);
        opt.optionString = jniOptsBuf;
        mOptions.add(opt);
```

```
    }

    if (stackTraceFileBuf[0] != '\0') {
        static const char* stfOptName = "-Xstacktracefile:";

        stackTraceFile = (char*) malloc(strlen(stfOptName) +
            strlen(stackTraceFileBuf) +1);
        strcpy(stackTraceFile, stfOptName);
        strcat(stackTraceFile, stackTraceFileBuf);
        opt.optionString = stackTraceFile;
        mOptions.add(opt);
    }

    /* extra options; parse this late so it overrides others */
    property_get("dalvik.vm.extra-opts", extraOptsBuf, "");
    parseExtraOpts(extraOptsBuf);

    /*设置本地属性*/
    {
        char langOption[sizeof("-Duser.language=") + 3];
        char regionOption[sizeof("-Duser.region=") + 3];
        strcpy(langOption, "-Duser.language=");
        strcpy(regionOption, "-Duser.region=");
        readLocale(langOption, regionOption);
        opt.extraInfo = NULL;
        opt.optionString = langOption;
        mOptions.add(opt);
        opt.optionString = regionOption;
        mOptions.add(opt);
    }
    opt.optionString = "-Djava.io.tmpdir=/sdcard";
    mOptions.add(opt);

    initArgs.version = JNI_VERSION_1_4;
    initArgs.options = mOptions.editArray();
    initArgs.nOptions = mOptions.size();
    initArgs.ignoreUnrecognized = JNI_FALSE;

    /*
     * 初始化 VM
     */
    if (JNI_CreateJavaVM(pJavaVM, pEnv, &initArgs) < 0) {
        ALOGE("JNI_CreateJavaVM failed\n");
        goto bail;
    }

    result = 0;

bail:
    free(stackTraceFile);
```

```
    return result;
}
```

由上述实现代码可知，函数 AndroidRuntime::startVm()最终会调用 JNI_CreateJavaVM()函数。此处的函数 JNI_CreateJavaVM()在文件 art/runtime/jni_internal.cc 中定义，具体实现代码如下所示。

```
extern "C" jint JNI_CreateJavaVM(JavaVM** p_vm, JNIEnv** p_env, void* vm_args) {
  const JavaVMInitArgs* args = static_cast<JavaVMInitArgs*>(vm_args);
  if (IsBadJniVersion(args->version)) {
    LOG(ERROR) << "Bad JNI version passed to CreateJavaVM: " << args->version;
    return JNI_EVERSION;
  }
  Runtime::Options options;
  for (int i = 0; i < args->nOptions; ++i) {
    JavaVMOption* option = &args->options[i];
    options.push_back(std::make_pair(std::string(option->optionString), option->extraInfo));
  }
  bool ignore_unrecognized = args->ignoreUnrecognized;
  if (!Runtime::Create(options, ignore_unrecognized)) {
    return JNI_ERR;
  }
  Runtime* runtime = Runtime::Current();
  bool started = runtime->Start();
  if (!started) {
    delete Thread::Current()->GetJniEnv();
    delete runtime->GetJavaVM();
    LOG(WARNING) << "CreateJavaVM failed";
    return JNI_ERR;
  }
  *p_env = Thread::Current()->GetJniEnv();
  *p_vm = runtime->GetJavaVM();
  return JNI_OK;
}
```

类 JniInvocation 的静态成员函数 GetJniInvocation()返回的便是前面所创建的 JniInvocation 实例。有了这个 JniInvocation 实例之后，就继续调用其成员函数 JNI_CreateJavaVM()来创建一个 JavaVM 接口及其对应的 JNIEnv 接口。

函数 GetJniInvocation()定义在文件 libnativehelper/JniInvocation.cpp 中，具体实现代码如下所示。

```
jint JniInvocation::JNI_CreateJavaVM(JavaVM** p_vm, JNIEnv** p_env, void* vm_args) {
  return JNI_CreateJavaVM_(p_vm, p_env, vm_args);
}
```

类 JniInvocation 的成员变量 JNI_CreateJavaVM_指向的就是前面所加载的 libdvm.so 或者 libart.so 所导出的函数 JNI_CreateJavaVM()。类 JniInvocation 的成员函数 JNI_CreateJavaVM()返回的 JavaVM 接口指向的要么是 Dalvik 虚拟机，要么是 ART 虚拟机。

10.1.3　创建运行实例

在文件 art/runtime/jni_internal.cc 中，函数 JNI_ CreateJavaVM()会调用函数 Create()创建 Runtime 的



实例。函数 Create()在文件 art/runtime/runtime.cc 中定义，具体实现代码如下所示。

```
bool Runtime::Create(const Options& options, bool ignore_unrecognized) {
  if (Runtime::instance_ != NULL) {
    return false;
  }
  InitLogging(NULL);   //初始化 Log 系统.
  instance_ = new Runtime; //创建 Runtime 实例
  if (!instance_->Init(options, ignore_unrecognized)) {
    delete instance_;
    instance_ = NULL;
    return false;//初始化 Runtime
  }
  return true;
}
```

再次回到函数 JNI_Create JavaVM()中，Runtime*runtime = Runtime::Current();用于获得 Runtime 当前实例，Runtime 使用单例模式实现。bool started = runtime->Start();用于调用 Start()函数，该函数在文件 art/runtime/runtime.cc 中定义，具体实现代码如下所示。

```
bool Runtime::Start() {
  VLOG(startup) << "Runtime::Start entering";
  CHECK(host_prefix_.empty()) << host_prefix_;
  Thread* self = Thread::Current();//获得当前运行线程
  self->TransitionFromRunnableToSuspended(kNative);//将该线程状态从 Runnable 切换到 Suspend

  started_ = true;
  //完成 Native 函数的初始化工作
  InitNativeMethods();

  InitThreadGroups(self);

  Thread::FinishStartup();

  if (is_zygote_) {
    if (!InitZygote()) {
      return false;
    }
  } else {
    DidForkFromZygote();
  }

  StartDaemonThreads();

  system_class_loader_ = CreateSystemClassLoader();

  self->GetJniEnv()->locals.AssertEmpty();

  VLOG(startup) << "Runtime::Start exiting";
```

```
  finished_starting_ = true;

  return true;
}
```

函数 Runtime::InitNativeMethods()在文件 art/runtime/runtime.cc 中定义，具体实现代码如下所示。

```
void Runtime::InitNativeMethods() {
  VLOG(startup) << "Runtime::InitNativeMethods entering";
  Thread* self = Thread::Current();
  JNIEnv* env = self->GetJniEnv();//获取 JNI 环境

  CHECK_EQ(self->GetState(), kNative);

  JniConstants::init(env);
  WellKnownClasses::Init(env);

  //调用 RegisterRuntimeNativeMethods()函数完成 Native 函数的注册
  RegisterRuntimeNativeMethods(env);

  {
    std::string mapped_name(StringPrintf(OS_SHARED_LIB_FORMAT_STR, "javacore"));
    std::string reason;
    self->TransitionFromSuspendedToRunnable();
    if (!instance_->java_vm_->LoadNativeLibrary(mapped_name, NULL, reason)) {
      LOG(FATAL) << "LoadNativeLibrary failed for \"" << mapped_name << "\": " << reason;
    }
    self->TransitionFromRunnableToSuspended(kNative);
  }

  WellKnownClasses::LateInit(env);

  VLOG(startup) << "Runtime::InitNativeMethods exiting";
}
```

10.1.4 注册本地 JNI 函数

在文件 art/runtime/runtime.cc 中的函数 Runtime::InitNativeMethods()中，通过代码行 RegisterRuntimeNativeMethods(env);调用函数 RegisterRuntimeNativeMethods()来注册 Native 函数，函数 RegisterRuntimeNativeMethods()的具体实现代码如下所示。

```
void Runtime::RegisterRuntimeNativeMethods(JNIEnv* env) {
#define REGISTER(FN) extern void FN(JNIEnv*); FN(env)
  REGISTER(register_java_lang_Throwable);
  REGISTER(register_dalvik_system_DexFile);
  REGISTER(register_dalvik_system_VMDebug);
  REGISTER(register_dalvik_system_VMRuntime);
  REGISTER(register_dalvik_system_VMStack);
  REGISTER(register_dalvik_system_Zygote);
  REGISTER(register_java_lang_Class);
  REGISTER(register_java_lang_DexCache);
```

```
  REGISTER(register_java_lang_Object);
  REGISTER(register_java_lang_Runtime);
  REGISTER(register_java_lang_String);
  REGISTER(register_java_lang_System);
  REGISTER(register_java_lang_Thread);
  REGISTER(register_java_lang_VMClassLoader);
  REGISTER(register_java_lang_reflect_Array);
  REGISTER(register_java_lang_reflect_Constructor);
  REGISTER(register_java_lang_reflect_Field);
  REGISTER(register_java_lang_reflect_Method);
  REGISTER(register_java_lang_reflect_Proxy);
  REGISTER(register_java_util_concurrent_atomic_AtomicLong);
  REGISTER(register_org_apache_harmony_dalvik_ddmc_DdmServer);
  REGISTER(register_org_apache_harmony_dalvik_ddmc_DdmVmInternal);
  REGISTER(register_sun_misc_Unsafe);
#undef REGISTER
}
```

在上述代码中列出了需要注册的函数列表，有关上述函数的具体实现请读者自行分析，例如 register_java_lang_Throwable 在文件 runtime/native/java_lang_Throwable.cc 中定义，具体实现代码如下所示。

```
void register_java_lang_Throwable(JNIEnv* env) {
  REGISTER_NATIVE_METHODS("java/lang/Throwable");
}
```

10.1.5　启动守护进程

再次返回到 Runtime::Start 函数，if (!InitZygote()) {代码行用于调用 InitZygote 完成一些文件系统的 mount 工作。然后通过 StartDaemonThreads();代码行调用 java.lang.Daemons.start()函数启动守护进程。函数 StartDaemonThreads()的具体实现代码如下所示。

```
void Runtime::StartDaemonThreads() {
  VLOG(startup) << "Runtime::StartDaemonThreads entering";

  Thread* self = Thread::Current();

  CHECK_EQ(self->GetState(), kNative);

  JNIEnv* env = self->GetJniEnv();
  env->CallStaticVoidMethod(WellKnownClasses::java_lang_Daemons,
                            WellKnownClasses::java_lang_Daemons_start);
  if (env->ExceptionCheck()) {
    env->ExceptionDescribe();
    LOG(FATAL) << "Error starting java.lang.Daemons";
  }

  VLOG(startup) << "Runtime::StartDaemonThreads exiting";
}
```

综上所述，Android 系统通过将 ART 运行时抽象成一个 Java 虚拟机，以及通过系统属性 persist.sys.

dalvik.vm.lib 和一个适配层 JniInvocation，就可以无缝地将 Dalvik 虚拟机替换为 ART 运行时。这个替换过程设计非常巧妙，因为涉及的代码修改是非常少的。涉及类的具体关系如图 10-2 所示。

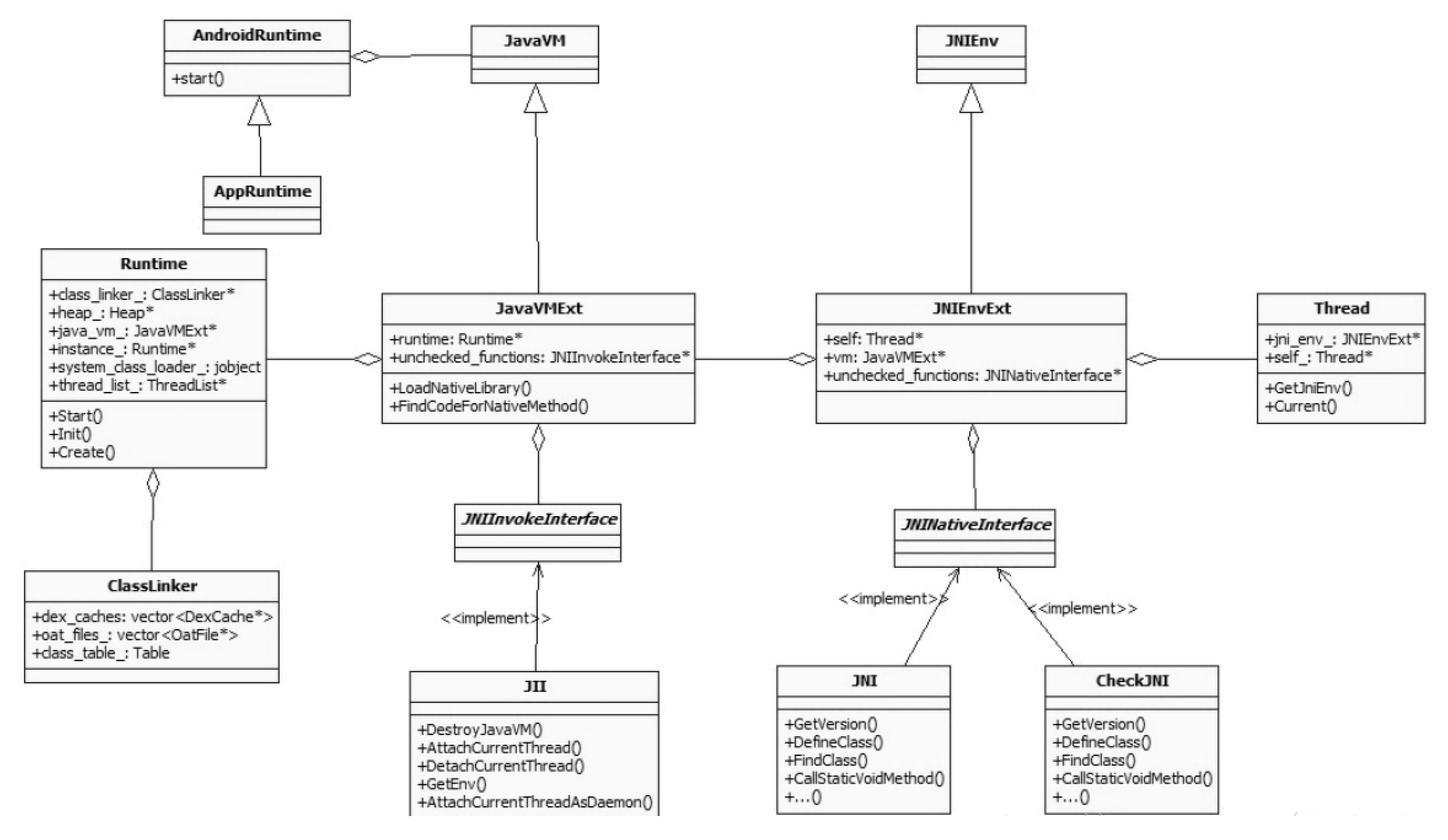


图 10-2　启动 ART 涉及的类

10.1.6　解析参数

在函数 JNI_CreateJavaVM()中，先调用 Create()函数创建 Runtime。Runtime 是一个单例，创建后会马上调用文件/art/runtime/runtime.cc 中的 Init()函数。Init()函数的功能是解析参数，初始化 Heap 和 JavaVMExt 结构，实现线程和信号处理，并创建 ClassLinker 等。函数 Init()的具体实现代码如下所示。

```
bool Runtime::Init(const RuntimeOptions& raw_options, bool ignore_unrecognized) {
  CHECK_EQ(sysconf(_SC_PAGE_SIZE), kPageSize);

  MemMap::Init();

  std::unique_ptr<ParsedOptions> options(ParsedOptions::Create(raw_options, ignore_unrecognized));
  if (options.get() == nullptr) {
    LOG(ERROR) << "Failed to parse options";
    return false;
  }
  VLOG(startup) << "Runtime::Init -verbose:startup enabled";

  QuasiAtomic::Startup();

  Monitor::Init(options->lock_profiling_threshold_, options->hook_is_sensitive_thread_);
```

```
boot_class_path_string_ = options->boot_class_path_string_;
class_path_string_ = options->class_path_string_;
properties_ = options->properties_;

compiler_callbacks_ = options->compiler_callbacks_;
patchoat_executable_ = options->patchoat_executable_;
must_relocate_ = options->must_relocate_;
is_zygote_ = options->is_zygote_;
is_explicit_gc_disabled_ = options->is_explicit_gc_disabled_;
dex2oat_enabled_ = options->dex2oat_enabled_;
image_dex2oat_enabled_ = options->image_dex2oat_enabled_;

vfprintf_ = options->hook_vfprintf_;
exit_ = options->hook_exit_;
abort_ = options->hook_abort_;

default_stack_size_ = options->stack_size_;
stack_trace_file_ = options->stack_trace_file_;

compiler_executable_ = options->compiler_executable_;
compiler_options_ = options->compiler_options_;
image_compiler_options_ = options->image_compiler_options_;
image_location_ = options->image_;

max_spins_before_thin_lock_inflation_ = options->max_spins_before_thin_lock_inflation_;

monitor_list_ = new MonitorList;
monitor_pool_ = MonitorPool::Create();
thread_list_ = new ThreadList;
intern_table_ = new InternTable;

verify_ = options->verify_;

if (options->interpreter_only_) {
  GetInstrumentation()->ForceInterpretOnly();
}

heap_ = new gc::Heap(options->heap_initial_size_,
                     options->heap_growth_limit_,
                     options->heap_min_free_,
                     options->heap_max_free_,
                     options->heap_target_utilization_,
                     options->foreground_heap_growth_multiplier_,
                     options->heap_maximum_size_,
                     options->heap_non_moving_space_capacity_,
                     options->image_,
                     options->image_isa_,
                     options->collector_type_,
                     options->background_collector_type_,
                     options->parallel_gc_threads_,
                     options->conc_gc_threads_,
```

```
                            options->low_memory_mode_,
                            options->long_pause_log_threshold_,
                            options->long_gc_log_threshold_,
                            options->ignore_max_footprint_,
                            options->use_tlab_,
                            options->verify_pre_gc_heap_,
                            options->verify_pre_sweeping_heap_,
                            options->verify_post_gc_heap_,
                            options->verify_pre_gc_rosalloc_,
                            options->verify_pre_sweeping_rosalloc_,
                            options->verify_post_gc_rosalloc_,
                            options->use_homogeneous_space_compaction_for_oom_,
                            options->min_interval_homogeneous_space_compaction_by_oom_);

dump_gc_performance_on_shutdown_ = options->dump_gc_performance_on_shutdown_;

BlockSignals();
InitPlatformSignalHandlers();

switch (kRuntimeISA) {
  case kArm:
  case kThumb2:
  case kX86:
  case kArm64:
  case kX86_64:
    implicit_null_checks_ = true;
    implicit_so_checks_ = true;
    break;
  default:
    break;
}

InitializeSignalChain();

if (implicit_null_checks_ || implicit_so_checks_ || implicit_suspend_checks_) {
  fault_manager.Init();

  if (implicit_suspend_checks_) {
    suspend_handler_ = new SuspensionHandler(&fault_manager);
  }

  if (implicit_so_checks_) {
    stack_overflow_handler_ = new StackOverflowHandler(&fault_manager);
  }

  if (implicit_null_checks_) {
    null_pointer_handler_ = new NullPointerHandler(&fault_manager);
  }

  if (kEnableJavaStackTraceHandler) {
    new JavaStackTraceHandler(&fault_manager);
```

```
    }
  }

  java_vm_ = new JavaVMExt(this, options.get());

  Thread::Startup();
…
```

在文件/art/runtime/runtime.cc 中，通过函数 Runtime::ParsedOptions*Runtime::ParsedOptions::Create()解析参数，将 raw_options 中的参数放入 parsed，如对环境变量 BOOTCLASSPATH 和 CLASSPATH 的处理。函数 Runtime::ParsedOptions*Runtime::ParsedOptions::Create()的具体实现代码如下所示。

```
Runtime::ParsedOptions* Runtime::ParsedOptions::Create(const Options& options, bool ignore_unrecognized) {
  UniquePtr<ParsedOptions> parsed(new ParsedOptions());
  const char* boot_class_path_string = getenv("BOOTCLASSPATH");
  if (boot_class_path_string != NULL) {
    parsed->boot_class_path_string_ = boot_class_path_string;
  }
  const char* class_path_string = getenv("CLASSPATH");
  if (class_path_string != NULL) {
    parsed->class_path_string_ = class_path_string;
  }
  parsed->check_jni_ = kIsDebugBuild;

  parsed->heap_initial_size_ = gc::Heap::kDefaultInitialSize;
  parsed->heap_maximum_size_ = gc::Heap::kDefaultMaximumSize;
  parsed->heap_min_free_ = gc::Heap::kDefaultMinFree;
  parsed->heap_max_free_ = gc::Heap::kDefaultMaxFree;
  parsed->heap_target_utilization_ = gc::Heap::kDefaultTargetUtilization;
  parsed->heap_growth_limit_ = 0;    // 0 means no growth limit
  parsed->parallel_gc_threads_ = sysconf(_SC_NPROCESSORS_CONF) - 1;
  parsed->conc_gc_threads_ = 0;
  parsed->stack_size_ = 0;    // 0 means default
  parsed->low_memory_mode_ = false;
……
```

初始化 Monitor（相当于 mutex+conditional variable，可用于多个线程同步）和线程链表等，然后实现比较重要的 Heap 及 GC 的初始化工作。其中，gc::Heap 功能通过文件 art/runtime/gc/heap.cc 中的函数 Heap::Heap()实现，具体实现代码如下所示。

```
Heap::Heap(size_t initial_size, size_t growth_limit, size_t min_free, size_t max_free,
           double target_utilization, size_t capacity, const std::string& original_image_file_name,
           bool concurrent_gc, size_t parallel_gc_threads, size_t conc_gc_threads,
           bool low_memory_mode, size_t long_pause_log_threshold, size_t long_gc_log_threshold,
           bool ignore_max_footprint)
    : alloc_space_(NULL),
      card_table_(NULL),
      concurrent_gc_(concurrent_gc),
      parallel_gc_threads_(parallel_gc_threads),
      conc_gc_threads_(conc_gc_threads),
      low_memory_mode_(low_memory_mode),
```

```
long_pause_log_threshold_(long_pause_log_threshold),
long_gc_log_threshold_(long_gc_log_threshold),
ignore_max_footprint_(ignore_max_footprint),
have_zygote_space_(false),
soft_ref_queue_lock_(NULL),
weak_ref_queue_lock_(NULL),
finalizer_ref_queue_lock_(NULL),
phantom_ref_queue_lock_(NULL),
is_gc_running_(false),
last_gc_type_(collector::kGcTypeNone),
next_gc_type_(collector::kGcTypePartial),
capacity_(capacity),
growth_limit_(growth_limit),
max_allowed_footprint_(initial_size),
native_footprint_gc_watermark_(initial_size),
native_footprint_limit_(2 * initial_size),
activity_thread_class_(NULL),
application_thread_class_(NULL),
activity_thread_(NULL),
application_thread_(NULL),
last_process_state_id_(NULL),
care_about_pause_times_(true),
concurrent_start_bytes_(concurrent_gc_ ? initial_size - kMinConcurrentRemainingBytes
    :   std::numeric_limits<size_t>::max()),
total_bytes_freed_ever_(0),
total_objects_freed_ever_(0),
large_object_threshold_(3 * kPageSize),
num_bytes_allocated_(0),
native_bytes_allocated_(0),
gc_memory_overhead_(0),
verify_missing_card_marks_(false),
verify_system_weaks_(false),
verify_pre_gc_heap_(false),
verify_post_gc_heap_(false),
verify_mod_union_table_(false),
min_alloc_space_size_for_sticky_gc_(2 * MB),
min_remaining_space_for_sticky_gc_(1 * MB),
last_trim_time_ms_(0),
allocation_rate_(0),
/* For GC a lot mode, we limit the allocations stacks to be kGcAlotInterval allocations. This
 * causes a lot of GC since we do a GC for alloc whenever the stack is full. When heap
 * verification is enabled, we limit the size of allocation stacks to speed up their
 * searching
 */
max_allocation_stack_size_(kGCALotMode ? kGcAlotInterval
    : (kDesiredHeapVerification > kNoHeapVerification) ? KB : MB),
reference_referent_offset_(0),
reference_queue_offset_(0),
reference_queueNext_offset_(0),
reference_pendingNext_offset_(0),
finalizer_reference_zombie_offset_(0),
```

```
        min_free_(min_free),
        max_free_(max_free),
        target_utilization_(target_utilization),
        total_wait_time_(0),
        total_allocation_time_(0),
        verify_object_mode_(kHeapVerificationNotPermitted),
        running_on_valgrind_(RUNNING_ON_VALGRIND) {
  if (VLOG_IS_ON(heap) || VLOG_IS_ON(startup)) {
    LOG(INFO) << "Heap() entering";
  }

  live_bitmap_.reset(new accounting::HeapBitmap(this));
  mark_bitmap_.reset(new accounting::HeapBitmap(this));

  byte* requested_alloc_space_begin = NULL;
  std::string image_file_name(original_image_file_name);
  if (!image_file_name.empty()) {
    space::ImageSpace* image_space = space::ImageSpace::Create(image_file_name);
    CHECK(image_space != NULL) << "Failed to create space for " << image_file_name;
    AddContinuousSpace(image_space);
    byte* oat_file_end_addr = image_space->GetImageHeader().GetOatFileEnd();
    CHECK_GT(oat_file_end_addr, image_space->End());
    if (oat_file_end_addr > requested_alloc_space_begin) {
      requested_alloc_space_begin =
          reinterpret_cast<byte*>(RoundUp(reinterpret_cast<uintptr_t>(oat_file_end_addr),
                                          kPageSize));
    }
  }

  alloc_space_ = space::DlMallocSpace::Create(Runtime::Current()->IsZygote() ? "zygote space" : "alloc
space",
                                              initial_size,
                                              growth_limit, capacity,
                                              requested_alloc_space_begin);
  CHECK(alloc_space_ != NULL) << "Failed to create alloc space";
  alloc_space_->SetFootprintLimit(alloc_space_->Capacity());
  AddContinuousSpace(alloc_space_);

  const bool kUseFreeListSpaceForLOS = false;
  if (kUseFreeListSpaceForLOS) {
    large_object_space_ = space::FreeListSpace::Create("large object space", NULL, capacity);
  } else {
    large_object_space_ = space::LargeObjectMapSpace::Create("large object space");
  }
  CHECK(large_object_space_ != NULL) << "Failed to create large object space";
  AddDiscontinuousSpace(large_object_space_);

  byte* heap_begin = continuous_spaces_.front()->Begin();
  size_t heap_capacity = continuous_spaces_.back()->End() - continuous_spaces_.front()->Begin();
  if (continuous_spaces_.back()->IsDlMallocSpace()) {
    heap_capacity += continuous_spaces_.back()->AsDlMallocSpace()->NonGrowthLimitCapacity();
```

```
    }

    card_table_.reset(accounting::CardTable::Create(heap_begin, heap_capacity));
    CHECK(card_table_.get() != NULL) << "Failed to create card table";

    image_mod_union_table_.reset(new accounting::ModUnionTableToZygoteAllocspace(this));
    CHECK(image_mod_union_table_.get() != NULL) << "Failed to create image mod-union table";

    zygote_mod_union_table_.reset(new accounting::ModUnionTableCardCache(this));
    CHECK(zygote_mod_union_table_.get() != NULL) << "Failed to create Zygote mod-union table";

    num_bytes_allocated_ = 0;

    static const size_t default_mark_stack_size = 64 * KB;
    mark_stack_.reset(accounting::ObjectStack::Create("mark stack", default_mark_stack_size));
    allocation_stack_.reset(accounting::ObjectStack::Create("allocation stack",
                                                            max_allocation_stack_size_));
    live_stack_.reset(accounting::ObjectStack::Create("live stack",
                                                      max_allocation_stack_size_));

    gc_complete_lock_ = new Mutex("GC complete lock");
    gc_complete_cond_.reset(new ConditionVariable("GC complete condition variable",
                                                  *gc_complete_lock_));

    soft_ref_queue_lock_ = new Mutex("Soft reference queue lock");
    weak_ref_queue_lock_ = new Mutex("Weak reference queue lock");
    finalizer_ref_queue_lock_ = new Mutex("Finalizer reference queue lock");
    phantom_ref_queue_lock_ = new Mutex("Phantom reference queue lock");

    last_gc_time_ns_ = NanoTime();
    last_gc_size_ = GetBytesAllocated();

    if (ignore_max_footprint_) {
      SetIdealFootprint(std::numeric_limits<size_t>::max());
      concurrent_start_bytes_ = max_allowed_footprint_;
    }

    for (size_t i = 0; i < 2; ++i) {
      const bool concurrent = i != 0;
      mark_sweep_collectors_.push_back(new collector::MarkSweep(this, concurrent));
      mark_sweep_collectors_.push_back(new collector::PartialMarkSweep(this, concurrent));
      mark_sweep_collectors_.push_back(new collector::StickyMarkSweep(this, concurrent));
    }

    CHECK_NE(max_allowed_footprint_, 0U);
    if (VLOG_IS_ON(heap) || VLOG_IS_ON(startup)) {
      LOG(INFO) << "Heap() exiting";
    }
}
```

在上述代码中，函数 ImageSpace::Create()会检测 image 文件，如果没有就调用 GenerateImage()来

创建。正因为上述操作过程，所以 log 中会有如下信息。

```
I/art      ( 161): GenerateImage: /system/bin/dex2oat--image=/data/dalvik-cache/system@framework@boot.art
--runtime-arg -Xms64m--runtime-arg -Xmx64m --dex-file=/system/framework/core-libart.jar ... --oat-file=/data/
dalvik-cache/system@framework@boot.oat--base=0x60000000--image-classes-zip=/system/framework/framework...
```

上述过程调用了 dex2oat，把 BOOTCLASSPATH 中的包打成 IMAGE 文件，最后会生成两个文件 boot.art 和 boot.oat。其中，前者是 IMAGE 文件，后者是 ELF 文件。这个 IMAGE 文件会被放到创建的 Heap 中。在函数 Heap::Heap()中，接下来会为一些数据结构分配空间，创建各种互斥量及初始化 GC。其中，MarkSweep、PartialMarkSweep 和 StickyMarkSweep 都是 art::gc::collector::GarbageCollector 的继承类，其几个子类应用了 Template Method 模式。在函数 GarbageCollector::Run()中实现了主要算法，此函数在文件 art/runtime/gc/collector/garbage_collector.cc 中定义，具体实现代码如下所示。

```
void GarbageCollector::Run() {
  ThreadList* thread_list = Runtime::Current()->GetThreadList();
  uint64_t start_time = NanoTime();
  pause_times_.clear();
  duration_ns_ = 0;

  InitializePhase();

  if (!IsConcurrent()) {
    // Pause is the entire length of the GC
    uint64_t pause_start = NanoTime();
    ATRACE_BEGIN("Application threads suspended");
    thread_list->SuspendAll();
    MarkingPhase();
    ReclaimPhase();
    thread_list->ResumeAll();
    ATRACE_END();
    uint64_t pause_end = NanoTime();
    pause_times_.push_back(pause_end - pause_start);
  } else {
    Thread* self = Thread::Current();
    {
      ReaderMutexLock mu(self, *Locks::mutator_lock_);
      MarkingPhase();
    }
    bool done = false;
    while (!done) {
      uint64_t pause_start = NanoTime();
      ATRACE_BEGIN("Suspending mutator threads");
      thread_list->SuspendAll();
      ATRACE_END();
      ATRACE_BEGIN("All mutator threads suspended");
      done = HandleDirtyObjectsPhase();
      ATRACE_END();
      uint64_t pause_end = NanoTime();
      ATRACE_BEGIN("Resuming mutator threads");
```

```
        thread_list->ResumeAll();
        ATRACE_END();
        pause_times_.push_back(pause_end - pause_start);
      }
      {
        ReaderMutexLock mu(self, *Locks::mutator_lock_);
        ReclaimPhase();
      }
    }

    uint64_t end_time = NanoTime();
    duration_ns_ = end_time - start_time;

    FinishPhase();
}
```

在上述代码中调用了 InitializePhase()、MarkingPhase()、ReclaimPhase()和 FinishPhase()等虚函数，这几个虚函数在 MarkSweep 等几个子类中有具体实现。

再次回到函数 Runtime::Init()，通过如下代码实现 ClassLinker 的初始化操作，其主要功能是调用 CreateFromImage()函数实现的。

```
ClassLinker* ClassLinker::CreateFromImage(InternTable* intern_table) {
    UniquePtr<ClassLinker> class_linker(new ClassLinker(intern_table));
    class_linker->InitFromImage();
    return class_linker.release();
}
```

在文件 art/art/runtime/class_linker.cc 中，通过函数 InitFromImage()从 Heap 中得到 image 的空间，然后得到 dex caches 数组，接着把这些 dex caches 对应的 dex file 信息注册到 BootClassPath 中去。函数 InitFromImage()的具体实现代码如下所示。

```
void ClassLinker::InitFromImage() {
    VLOG(startup) << "ClassLinker::InitFromImage entering";
    CHECK(!init_done_);

    gc::Heap* heap = Runtime::Current()->GetHeap();
    gc::space::ImageSpace* space = heap->GetImageSpace();
    dex_cache_image_class_lookup_required_ = true;
    CHECK(space != NULL);
    OatFile& oat_file = GetImageOatFile(space);
    CHECK_EQ(oat_file.GetOatHeader().GetImageFileLocationOatChecksum(), 0U);
    CHECK_EQ(oat_file.GetOatHeader().GetImageFileLocationOatDataBegin(), 0U);
    CHECK(oat_file.GetOatHeader().GetImageFileLocation().empty());
    portable_resolution_trampoline_ = oat_file.GetOatHeader().GetPortableResolutionTrampoline();
    quick_resolution_trampoline_ = oat_file.GetOatHeader().GetQuickResolutionTrampoline();
    mirror::Object* dex_caches_object = space->GetImageHeader().GetImageRoot(ImageHeader::kDexCaches);
    mirror::ObjectArray<mirror::DexCache>* dex_caches =
        dex_caches_object->AsObjectArray<mirror::DexCache>();
```

```
mirror::ObjectArray<mirror::Class>* class_roots =
    space->GetImageHeader().GetImageRoot(ImageHeader::kClassRoots)->AsObjectArray<mirror::Class>();
class_roots_ = class_roots;

mirror::String::SetClass(GetClassRoot(kJavaLangString));

CHECK_EQ(oat_file.GetOatHeader().GetDexFileCount(),
         static_cast<uint32_t>(dex_caches->GetLength()));
Thread* self = Thread::Current();
for (int32_t i = 0; i < dex_caches->GetLength(); i++) {
  SirtRef<mirror::DexCache> dex_cache(self, dex_caches->Get(i));
  const std::string& dex_file_location(dex_cache->GetLocation()->ToModifiedUtf8());
  const OatFile::OatDexFile* oat_dex_file = oat_file.GetOatDexFile(dex_file_location, NULL);
  CHECK(oat_dex_file != NULL) << oat_file.GetLocation() << " " << dex_file_location;
  const DexFile* dex_file = oat_dex_file->OpenDexFile();
  if (dex_file == NULL) {
    LOG(FATAL) << "Failed to open dex file " << dex_file_location
               << " from within oat file " << oat_file.GetLocation();
  }

  CHECK_EQ(dex_file->GetLocationChecksum(), oat_dex_file->GetDexFileLocationChecksum());

  AppendToBootClassPath(*dex_file, dex_cache);
}

mirror::ArtMethod::SetClass(GetClassRoot(kJavaLangReflectArtMethod));

if (Runtime::Current()->GetInstrumentation()->InterpretOnly()) {
  ReaderMutexLock mu(self, *Locks::heap_bitmap_lock_);
  heap->FlushAllocStack();
  heap->GetLiveBitmap()->Walk(InitFromImageInterpretOnlyCallback, this);
}

mirror::Class::SetClassClass(class_roots->Get(kJavaLangClass));
class_roots_ = class_roots;

array_iftable_ = GetClassRoot(kObjectArrayClass)->GetIfTable();
DCHECK(array_iftable_ == GetClassRoot(kBooleanArrayClass)->GetIfTable());
mirror::ArtField::SetClass(GetClassRoot(kJavaLangReflectArtField));
mirror::BooleanArray::SetArrayClass(GetClassRoot(kBooleanArrayClass));
mirror::ByteArray::SetArrayClass(GetClassRoot(kByteArrayClass));
mirror::CharArray::SetArrayClass(GetClassRoot(kCharArrayClass));
mirror::DoubleArray::SetArrayClass(GetClassRoot(kDoubleArrayClass));
mirror::FloatArray::SetArrayClass(GetClassRoot(kFloatArrayClass));
mirror::IntArray::SetArrayClass(GetClassRoot(kIntArrayClass));
mirror::LongArray::SetArrayClass(GetClassRoot(kLongArrayClass));
mirror::ShortArray::SetArrayClass(GetClassRoot(kShortArrayClass));
mirror::Throwable::SetClass(GetClassRoot(kJavaLangThrowable));
mirror::StackTraceElement::SetClass(GetClassRoot(kJavaLangStackTraceElement));
```

```
    FinishInit();

    VLOG(startup) << "ClassLinker::InitFromImage exiting";
}
```

在文件 art/art/runtime/class_linker.cc 中，通过函数 AppendToBootClassPath()和 RegisterDexFileLocked()将 dex cache 和 dex file 关联起来，同时把 dex file 注册到 boot_class_path_，将 dex cache 注册到 dex_caches_。函数 AppendToBootClassPath()和 RegisterDexFileLocked()的具体实现代码如下所示。

```
void ClassLinker::AppendToBootClassPath(const DexFile& dex_file, SirtRef<mirror::DexCache>& dex_cache) {
    CHECK(dex_cache.get() != NULL) << dex_file.GetLocation();
    boot_class_path_.push_back(&dex_file);
    RegisterDexFile(dex_file, dex_cache);
}
void ClassLinker::RegisterDexFileLocked(const DexFile& dex_file, SirtRef<mirror::DexCache>& dex_cache) {
    dex_lock_.AssertExclusiveHeld(Thread::Current());
    CHECK(dex_cache.get() != NULL) << dex_file.GetLocation();
    CHECK(dex_cache->GetLocation()->Equals(dex_file.GetLocation()))
        << dex_cache->GetLocation()->ToModifiedUtf8() << " " << dex_file.GetLocation();
    dex_caches_.push_back(dex_cache.get());
    dex_cache->SetDexFile(&dex_file);
    dex_caches_dirty_ = true;
}
```

当注册上述信息后，在 ClassLinker 调用 FindClass()函数时会用到。执行完 Runtime 中的函数 Create()和 Init()后，在 JNI_CreateJavaVM()函数中 Runtime 的 Start()函数会被调用。

10.1.7 初始化类、方法和域

在文件 runtime.cc 的函数 InitNativeMethods()中分别调用函数 JniConstants::init()和 WellKnownClasses::Init()。函数 InitNativeMethods()的具体实现代码如下所示。

```
void Runtime::InitNativeMethods() {
    VLOG(startup) << "Runtime::InitNativeMethods entering";
    Thread* self = Thread::Current();
    JNIEnv* env = self->GetJniEnv();

    CHECK_EQ(self->GetState(), kNative);

    JniConstants::init(env);
    WellKnownClasses::Init(env);

    RegisterRuntimeNativeMethods(env);

    {
        std::string mapped_name(StringPrintf(OS_SHARED_LIB_FORMAT_STR, "javacore"));
        std::string reason;
        self->TransitionFromSuspendedToRunnable();
        if (!instance_->java_vm_->LoadNativeLibrary(mapped_name, NULL, reason)) {
            LOG(FATAL) << "LoadNativeLibrary failed for \"" << mapped_name << "\": " << reason;
        }
        self->TransitionFromRunnableToSuspended(kNative);
```

```
    }

    WellKnownClasses::LateInit(env);

    VLOG(startup) << "Runtime::InitNativeMethods exiting";
}
```

函数 JniConstants::init()在文件 libnativehelper/JniConstants.cpp 中定义，WellKnownClasses::Init()在文件 art/runtime/well_known_classes.cc 中定义，这两个函数的具体实现代码如下所示。

```
void JniConstants::init(JNIEnv* env) {
    bidiRunClass = findClass(env, "java/text/Bidi$Run");
    bigDecimalClass = findClass(env, "java/math/BigDecimal");
    booleanClass = findClass(env, "java/lang/Boolean");
    byteClass = findClass(env, "java/lang/Byte");
    byteArrayClass = findClass(env, "[B");
    calendarClass = findClass(env, "java/util/Calendar");
    characterClass = findClass(env, "java/lang/Character");
    charsetICUClass = findClass(env, "java/nio/charset/CharsetICU");
    constructorClass = findClass(env, "java/lang/reflect/Constructor");
    floatClass = findClass(env, "java/lang/Float");
    deflaterClass = findClass(env, "java/util/zip/Deflater");
    doubleClass = findClass(env, "java/lang/Double");
    errnoExceptionClass = findClass(env, "libcore/io/ErrnoException");
    fieldClass = findClass(env, "java/lang/reflect/Field");
    fieldPositionIteratorClass = findClass(env, "libcore/icu/NativeDecimalFormat$FieldPositionIterator");
    fileDescriptorClass = findClass(env, "java/io/FileDescriptor");
    gaiExceptionClass = findClass(env, "libcore/io/GaiException");
    inet6AddressClass = findClass(env, "java/net/Inet6Address");
    inetAddressClass = findClass(env, "java/net/InetAddress");
    inetSocketAddressClass = findClass(env, "java/net/InetSocketAddress");
    inetUnixAddressClass = findClass(env, "java/net/InetUnixAddress");
    inflaterClass = findClass(env, "java/util/zip/Inflater");
    inputStreamClass = findClass(env, "java/io/InputStream");
    integerClass = findClass(env, "java/lang/Integer");
    localeDataClass = findClass(env, "libcore/icu/LocaleData");
    longClass = findClass(env, "java/lang/Long");
    methodClass = findClass(env, "java/lang/reflect/Method");
    mutableIntClass = findClass(env, "libcore/util/MutableInt");
    mutableLongClass = findClass(env, "libcore/util/MutableLong");
    objectClass = findClass(env, "java/lang/Object");
    objectArrayClass = findClass(env, "[Ljava/lang/Object;");
    outputStreamClass = findClass(env, "java/io/OutputStream");
    parsePositionClass = findClass(env, "java/text/ParsePosition");
    patternSyntaxExceptionClass = findClass(env, "java/util/regex/PatternSyntaxException");
    realToStringClass = findClass(env, "java/lang/RealToString");
    referenceClass = findClass(env, "java/lang/ref/Reference");
    shortClass = findClass(env, "java/lang/Short");
    socketClass = findClass(env, "java/net/Socket");
    socketImplClass = findClass(env, "java/net/SocketImpl");
    stringClass = findClass(env, "java/lang/String");
    structAddrinfoClass = findClass(env, "libcore/io/StructAddrinfo");
    structFlockClass = findClass(env, "libcore/io/StructFlock");
```

```
    structGroupReqClass = findClass(env, "libcore/io/StructGroupReq");
    structLingerClass = findClass(env, "libcore/io/StructLinger");
    structPasswdClass = findClass(env, "libcore/io/StructPasswd");
    structPollfdClass = findClass(env, "libcore/io/StructPollfd");
    structStatClass = findClass(env, "libcore/io/StructStat");
    structStatVfsClass = findClass(env, "libcore/io/StructStatVfs");
    structTimevalClass = findClass(env, "libcore/io/StructTimeval");
    structUcredClass = findClass(env, "libcore/io/StructUcred");
    structUtsnameClass = findClass(env, "libcore/io/StructUtsname");
}
void WellKnownClasses::Init(JNIEnv* env) {
  com_android_dex_Dex = CacheClass(env, "com/android/dex/Dex");
  dalvik_system_PathClassLoader = CacheClass(env, "dalvik/system/PathClassLoader");
  java_lang_ClassLoader = CacheClass(env, "java/lang/ClassLoader");
  java_lang_ClassNotFoundException = CacheClass(env, "java/lang/ClassNotFoundException");
  java_lang_Daemons = CacheClass(env, "java/lang/Daemons");
  java_lang_Object = CacheClass(env, "java/lang/Object");
  java_lang_Error = CacheClass(env, "java/lang/Error");
  java_lang_reflect_AbstractMethod = CacheClass(env, "java/lang/reflect/AbstractMethod");
  java_lang_reflect_ArtMethod = CacheClass(env, "java/lang/reflect/ArtMethod");
  java_lang_reflect_Constructor = CacheClass(env, "java/lang/reflect/Constructor");
  java_lang_reflect_Field = CacheClass(env, "java/lang/reflect/Field");
  java_lang_reflect_Method = CacheClass(env, "java/lang/reflect/Method");
  java_lang_reflect_Proxy = CacheClass(env, "java/lang/reflect/Proxy");
  java_lang_RuntimeException = CacheClass(env, "java/lang/RuntimeException");
  java_lang_StackOverflowError = CacheClass(env, "java/lang/StackOverflowError");
  java_lang_System = CacheClass(env, "java/lang/System");
  java_lang_Thread = CacheClass(env, "java/lang/Thread");
  java_lang_Thread$UncaughtExceptionHandler = CacheClass(env, "java/lang/Thread$UncaughtException
Handler");
  java_lang_ThreadGroup = CacheClass(env, "java/lang/ThreadGroup");
  java_lang_Throwable = CacheClass(env, "java/lang/Throwable");
  java_nio_DirectByteBuffer = CacheClass(env, "java/nio/DirectByteBuffer");
  org_apache_harmony_dalvik_ddmc_Chunk = CacheClass(env, "org/apache/harmony/dalvik/ddmc/Chunk");
  org_apache_harmony_dalvik_ddmc_DdmServer = CacheClass(env, "org/apache/harmony/dalvik/ddmc/
DdmServer");

  com_android_dex_Dex_create = CacheMethod(env, com_android_dex_Dex, true, "create", "(Ljava/nio/
ByteBuffer;)Lcom/android/dex/Dex;");
  java_lang_ClassNotFoundException_init = CacheMethod(env, java_lang_ClassNotFoundException, false,
"<init>", "(Ljava/lang/String;Ljava/lang/Throwable;)V");
  java_lang_ClassLoader_loadClass = CacheMethod(env, java_lang_ClassLoader, false, "loadClass", "(Ljava/
lang/String;)Ljava/lang/Class;");

  java_lang_Daemons_requestGC = CacheMethod(env, java_lang_Daemons, true, "requestGC", "()V");
  java_lang_Daemons_requestHeapTrim = CacheMethod(env, java_lang_Daemons, true, "requestHeapTrim",
"()V");
  java_lang_Daemons_start = CacheMethod(env, java_lang_Daemons, true, "start", "()V");

  ScopedLocalRef<jclass> java_lang_ref_FinalizerReference(env, env->FindClass("java/lang/ref/FinalizerReference"));
  java_lang_ref_FinalizerReference_add = CacheMethod(env, java_lang_ref_FinalizerReference.get(), true,
"add", "(Ljava/lang/Object;)V");
```

```
ScopedLocalRef<jclass> java_lang_ref_ReferenceQueue(env, env->FindClass("java/lang/ref/ReferenceQueue"));
java_lang_ref_ReferenceQueue_add = CacheMethod(env, java_lang_ref_ReferenceQueue.get(), true, "add",
"(Ljava/lang/ref/Reference;)V");

java_lang_reflect_Proxy_invoke = CacheMethod(env, java_lang_reflect_Proxy, true, "invoke", "(Ljava/lang/
reflect/Proxy;Ljava/lang/reflect/ArtMethod;[Ljava/lang/Object;)Ljava/lang/Object;");
java_lang_Thread_init = CacheMethod(env, java_lang_Thread, false, "<init>", "(Ljava/lang/ThreadGroup;
Ljava/lang/String;IZ)V");
java_lang_Thread_run = CacheMethod(env, java_lang_Thread, false, "run", "()V");
java_lang_Thread$UncaughtExceptionHandler_uncaughtException = CacheMethod(env, java_lang_
Thread$ UncaughtExceptionHandler, false, "uncaughtException", "(Ljava/lang/Thread;Ljava/lang/
Throwable;)V");
java_lang_ThreadGroup_removeThread = CacheMethod(env, java_lang_ThreadGroup, false, "removeThread",
"(Ljava/lang/Thread;)V");
java_nio_DirectByteBuffer_init = CacheMethod(env, java_nio_DirectByteBuffer, false, "<init>", "(JI)V");
org_apache_harmony_dalvik_ddmc_DdmServer_broadcast = CacheMethod(env, org_apache_harmony_dalvik_
ddmc_DdmServer, true, "broadcast", "(I)V");
org_apache_harmony_dalvik_ddmc_DdmServer_dispatch = CacheMethod(env, org_apache_harmony_dalvik_
ddmc_DdmServer, true, "dispatch", "(I[BII)Lorg/apache/harmony/dalvik/ddmc/Chunk;");

java_lang_Thread_daemon = CacheField(env, java_lang_Thread, false, "daemon", "Z");
java_lang_Thread_group = CacheField(env, java_lang_Thread, false, "group", "Ljava/lang/ThreadGroup;");
java_lang_Thread_lock = CacheField(env, java_lang_Thread, false, "lock", "Ljava/lang/Object;");
java_lang_Thread_name = CacheField(env, java_lang_Thread, false, "name", "Ljava/lang/String;");
java_lang_Thread_priority = CacheField(env, java_lang_Thread, false, "priority", "I");
java_lang_Thread_uncaughtHandler = CacheField(env, java_lang_Thread, false, "uncaughtHandler", "Ljava/
lang/Thread$UncaughtExceptionHandler;");
java_lang_Thread_nativePeer = CacheField(env, java_lang_Thread, false, "nativePeer", "I");
java_lang_ThreadGroup_mainThreadGroup = CacheField(env, java_lang_ThreadGroup, true, "mainThreadGroup",
"Ljava/lang/ThreadGroup;");
java_lang_ThreadGroup_name = CacheField(env, java_lang_ThreadGroup, false, "name", "Ljava/lang/String;");
java_lang_ThreadGroup_systemThreadGroup = CacheField(env, java_lang_ThreadGroup, true, "system
ThreadGroup", "Ljava/lang/ThreadGroup;");
java_lang_reflect_AbstractMethod_artMethod = CacheField(env, java_lang_reflect_AbstractMethod, false,
"artMethod", "Ljava/lang/reflect/ArtMethod;");
java_lang_reflect_Field_artField = CacheField(env, java_lang_reflect_Field, false, "artField", "Ljava/lang/
reflect/ArtField;");
java_lang_reflect_Proxy_h = CacheField(env, java_lang_reflect_Proxy, false, "h", "Ljava/lang/reflect/
InvocationHandler;");
java_nio_DirectByteBuffer_capacity = CacheField(env, java_nio_DirectByteBuffer, false, "capacity", "I");
java_nio_DirectByteBuffer_effectiveDirectAddress = CacheField(env, java_nio_DirectByteBuffer, false,
"effectiveDirectAddress", "J");
org_apache_harmony_dalvik_ddmc_Chunk_data = CacheField(env, org_apache_harmony_dalvik_ddmc_
Chunk, false, "data", "[B");
org_apache_harmony_dalvik_ddmc_Chunk_length = CacheField(env, org_apache_harmony_dalvik_ddmc_
Chunk, false, "length", "I");
org_apache_harmony_dalvik_ddmc_Chunk_offset = CacheField(env, org_apache_harmony_dalvik_ddmc_
Chunk, false, "offset", "I");
org_apache_harmony_dalvik_ddmc_Chunk_type = CacheField(env, org_apache_harmony_dalvik_ddmc_
Chunk, false, "type", "I");
```

```
    java_lang_Boolean_valueOf = CachePrimitiveBoxingMethod(env, 'Z', "java/lang/Boolean");
    java_lang_Byte_valueOf = CachePrimitiveBoxingMethod(env, 'B', "java/lang/Byte");
    java_lang_Character_valueOf = CachePrimitiveBoxingMethod(env, 'C', "java/lang/Character");
    java_lang_Double_valueOf = CachePrimitiveBoxingMethod(env, 'D', "java/lang/Double");
    java_lang_Float_valueOf = CachePrimitiveBoxingMethod(env, 'F', "java/lang/Float");
    java_lang_Integer_valueOf = CachePrimitiveBoxingMethod(env, 'I', "java/lang/Integer");
    java_lang_Long_valueOf = CachePrimitiveBoxingMethod(env, 'J', "java/lang/Long");
    java_lang_Short_valueOf = CachePrimitiveBoxingMethod(env, 'S', "java/lang/Short");
}
```

通过上述代码可知，分别通过 FindClass()、GetStaticFieldID()和 GetStaticMethodID()等函数初始化了系统基本类、方法和域，这些都是最基本的类。

然后 RegisterRuntimeNativeMethods()函数注册了系统类中的 Native 函数。

```
void Runtime::RegisterRuntimeNativeMethods(JNIEnv* env) {
#define REGISTER(FN) extern void FN(JNIEnv*); FN(env)
    REGISTER(register_java_lang_Throwable);
    ...
```

接着函数 InitNativeMethods()会载入 libjavacore.so 库，单独载入是因为该库本身包含了 System.loadLibrary()实现，不先载入会导致顺序紊乱。

```
if (!instance_->java_vm_->LoadNativeLibrary(mapped_name, NULL, reason)) {
```

再次回到 Runtime::Start()函数进行线程初始化，再判断是否为 Zygote 进程。如果是则调用 InitZygote()进行初始化（其中主要是 mount 一些文件系统）操作，否则调用 DidForkFromZygote()函数。函数 DidForkFromZygote()会创建线程池，创建 signalcatcher 线程和启动 JDWP 调试线程）。此函数的主要作用是调用 Heap 对象的 CreateThreadPool()函数来创建线程池。函数 DidForkFromZygote()在文件 Runtime:.cc 中定义，具体实现代码如下所示。

```
void Runtime::DidForkFromZygote() {
    is_zygote_ = false;

    heap_->CreateThreadPool();

    StartSignalCatcher();

    Dbg::StartJdwp();
}
```

最后，Start()函数中调用了 StartDaemonThreads()函数，此函数的作用是调用 Java 类 Daemons 的 start()方法来启动一些 Deamon 线程，这个过程实际上和 Dalvik 启动时完成的最后一项工作相同。函数 Runtime::StartDaemonThreads()在文件 Runtime.cc 中定义，具体实现代码如下所示。

```
void Runtime::StartDaemonThreads() {
    VLOG(startup) << "Runtime::StartDaemonThreads entering";

    Thread* self = Thread::Current();

    CHECK_EQ(self->GetState(), kNative);

    JNIEnv* env = self->GetJniEnv();
```

```
  env->CallStaticVoidMethod(WellKnownClasses::java_lang_Daemons,
                            WellKnownClasses::java_lang_Daemons_start);
  if (env->ExceptionCheck()) {
    env->ExceptionDescribe();
    LOG(FATAL) << "Error starting java.lang.Daemons";
  }

  VLOG(startup) << "Runtime::StartDaemonThreads exiting";
}
```

启动后台线程，然后用 JNI 调用 java.lang.ClassLoader.getSystemClassLoader()得到系统的 ClassLoader（由 createSystemClassLoader()创建），稍后调用 com.android.internal.os.ZygoteInit.main()时用的就是 ClassLoader。

```
StartDaemonThreads();
system_class_loader_ = CreateSystemClassLoader();
```

从 StartVM()返回后，AndroidRuntime 执行 startReg()在创建线程时加一个 hook()函数，这样每个 Thread 启动时会先去执行 AndroidRuntime::javaThreadShell()，而该函数会初始化 Java 虚拟机环境，这样新建的线程就可以调用 Java 层了。函数 startReg()在文件 AndroidRuntime.cpp 中定义，具体实现代码如下所示。

```
int AndroidRuntime::startReg(JNIEnv* env)
{
    /*
     * This hook causes all future threads created in this process to be
     * attached to the JavaVM.  (This needs to go away in favor of JNI
     * Attach calls)
     */
    androidSetCreateThreadFunc((android_create_thread_fn) javaCreateThreadEtc);

    ALOGV("--- registering native functions ---\n");

    /*
     * Every "register" function calls one or more things that return
     * a local reference (e.g. FindClass).  Because we haven't really
     * started the VM yet, they're all getting stored in the base frame
     * and never released.  Use Push/Pop to manage the storage.
     */
    env->PushLocalFrame(200);

    if (register_jni_procs(gRegJNI, NELEM(gRegJNI), env) < 0) {
        env->PopLocalFrame(NULL);
        return -1;
    }
    env->PopLocalFrame(NULL);

    //createJavaThread("fubar", quickTest, (void*) "hello");

    return 0;
}

AndroidRuntime* AndroidRuntime::getRuntime()
```

```
{
    return gCurRuntime;
}
```

到此为止，AndroidRuntime 中的 start()函数的执行过程全部讲解完毕。总结的执行流程如图 10-3 所示。

sd libart.so
app_process
AndroidRuntime
JniInvocation
jni_internal
Runtime
gc Heap
JavaVMExt
Thread
JNIEnvExt
ClassLinker
<<create>>
1 : start()
<<create>>
2
3 : init()
4 : startVM()
5 : JNI_CreateJavaVM()
6 : JNI_CreateJavaVM()
7 : Create()
8 : Init()
<<create>>
10
9 : startReg()
11 : Heap
12 <<create>>
13 : JavaVMExt
14 : Startup()
15 : Attach()
<<create>>
16
17 : Init()
18 <<create>>
19 : JNIEnvExt
20 : CreateFromImage()
21 : InitFromImage()
22 : ClassLinker
JniConstants
WellKnownClasses
23 : Start()
24 : InitNativeMethods()
25 : init()
26 : init()
27 : RegisterRuntimeNativeMethods()
28 : InitThreadGroups()
29 : InitZygote()
30 : CreateSystemClassLoader()

图 10-3　函数 start()的执行流程

10.2　进入 main()主函数

完成 ART 的基本初始化工作后，接下来开始执行主函数，具体步骤如下所示。

☑　先通过 FindClass()找到相应的类。

☑　然后通过 GetStaticMethodID()找到相应的方法。

☑　最后调用 CallStaticVoidMethod()进入 Java 执行托管代码工作。

本节将以 Zygote 初始化操作为例进行讲解，其中，类名为 com.android.internal.os. ZygoteInit，方法为 main。并详细分析执行 ART 主函数的具体过程，为读者学习本书后面的知识打下基础。

在文件 frameworks/base/core/jni/AndroidRuntime.cpp 中，通过 AndroidRuntime::start 中如下代码调用 startVM()启动虚拟机，然后调用 startReg()注册 JNI 方法，并调用 com.android.internal.os.ZygoteInit 类的 main()函数。

```
    /*
     * Start VM.   This thread becomes the main thread of the VM, and will
     * not return until the VM exits
     */
    char* slashClassName = toSlashClassName(className);
    jclass startClass = env->FindClass(slashClassName);
    if (startClass == NULL) {
        ALOGE("JavaVM unable to locate class '%s'\n", slashClassName);
        /* keep going */
    } else {
        jmethodID startMeth = env->GetStaticMethodID(startClass, "main",
            "([Ljava/lang/String;)V");
        if (startMeth == NULL) {
            ALOGE("JavaVM unable to find main() in '%s'\n", className);
            /* keep going */
        } else {
            env->CallStaticVoidMethod(startClass, startMeth, strArray);

#if 0
            if (env->ExceptionCheck())
                threadExitUncaughtException(env);
#endif
        }
    }
```

在上述代码中涉及了 3 个函数：FindClass()、GetStaticMethodID()和 CallStaticVoidMethod()。函数 FindClass()在文件/art/runtime/jni_internal.cc 中实现，具体实现代码如下所示。

```
static jclass FindClass(JNIEnv* env, const char* name) {
    CHECK_NON_NULL_ARGUMENT(FindClass, name);
```

```
    Runtime* runtime = Runtime::Current();
    ClassLinker* class_linker = runtime->GetClassLinker();
    std::string descriptor(NormalizeJniClassDescriptor(name));
    ScopedObjectAccess soa(env);
    Class* c = NULL;
    if (runtime->IsStarted()) {
      ClassLoader* cl = GetClassLoader(soa);
      c = class_linker->FindClass(descriptor.c_str(), cl);
    } else {
      c = class_linker->FindSystemClass(descriptor.c_str());
    }
    return soa.AddLocalReference<jclass>(c);
}
```

函数 GetClassLoader()也是在文件 art/runtime/jni_internal.cc 中定义，功能是调用 GetSystemClassLoader()得到前面初始化好的系统 ClassLoader，具体实现代码如下所示。

```
static ClassLoader* GetClassLoader(const ScopedObjectAccess& soa)
    SHARED_LOCKS_REQUIRED(Locks::mutator_lock_) {
  ArtMethod* method = soa.Self()->GetCurrentMethod(NULL);
  if (method == soa.DecodeMethod(WellKnownClasses::java_lang_Runtime_nativeLoad)) {
    return soa.Self()->GetClassLoaderOverride();
  }
  if (method != NULL) {
    return method->GetDeclaringClass()->GetClassLoader();
  }
  ClassLoader* class_loader = soa.Decode<ClassLoader*>(Runtime::Current()->GetSystemClassLoader());
  if (class_loader != NULL) {
    return class_loader;
  }
  class_loader = soa.Self()->GetClassLoaderOverride();
  if (class_loader != NULL) {
    CHECK(Runtime::Current()->UseCompileTimeClassPath());
    return class_loader;
  }
  return NULL;
}
```

10.3 查找目标类

开始调用 ClassLinker 的函数 FindClass()查找目标类，这一过程中涉及的关键函数有 LookupClass()、DefineClass()、InsertClass()、LoadClass()和 LinkClass()，上述关键函数的具体说明如下。

10.3.1　函数 LookupClass()

函数 LookupClass()在文件 art/runtime/class_linker.cc 中定义，先在 ClassLinker 的成员变量 class_table_中寻找指定类，找到则返回，若找不到，则看是否要在 image 中查找（class_loader 为 NULL 且 dex_cache_image_class_lookup_required 为 true）。如果需要，则调用 LookupClassFromImage()在 Image 中进行查找，找到后，调用 InsertClass()将找到的类放入到 class_table_中以便下次查找。

函数 LookupClass()的具体实现代码如下所示。

```
mirror::Class* ClassLinker::FindClass(const char* descriptor, mirror::ClassLoader* class_loader) {
  DCHECK_NE(*descriptor, '\0') << "descriptor is empty string";
  Thread* self = Thread::Current();
  DCHECK(self != NULL);
  self->AssertNoPendingException();
  if (descriptor[1] == '\0') {
    return FindPrimitiveClass(descriptor[0]);
  }
  mirror::Class* klass = LookupClass(descriptor, class_loader);
  if (klass != NULL) {
    return EnsureResolved(self, klass);
  }
  if (descriptor[0] == '[') {
    return CreateArrayClass(descriptor, class_loader);

  } else if (class_loader == NULL) {
    DexFile::ClassPathEntry pair = DexFile::FindInClassPath(descriptor, boot_class_path_);
    if (pair.second != NULL) {
      return DefineClass(descriptor, NULL, *pair.first, *pair.second);
    }

  } else if (Runtime::Current()->UseCompileTimeClassPath()) {
    if (IsInBootClassPath(descriptor)) {
      mirror::Class* system_class = FindSystemClass(descriptor);
      CHECK(system_class != NULL);
      return system_class;
    }
    const std::vector<const DexFile*>* class_path;
    {
      ScopedObjectAccessUnchecked soa(self);
      ScopedLocalRef<jobject> jclass_loader(soa.Env(), soa.AddLocalReference<jobject>(class_loader));
      class_path = &Runtime::Current()->GetCompileTimeClassPath(jclass_loader.get());
    }

    DexFile::ClassPathEntry pair = DexFile::FindInClassPath(descriptor, *class_path);
    if (pair.second != NULL) {
      return DefineClass(descriptor, class_loader, *pair.first, *pair.second);
```

```
    }

  } else {
    ScopedObjectAccessUnchecked soa(self->GetJniEnv());
    ScopedLocalRef<jobject> class_loader_object(soa.Env(),
                                              soa.AddLocalReference<jobject>(class_loader));
    std::string class_name_string(DescriptorToDot(descriptor));
    ScopedLocalRef<jobject> result(soa.Env(), NULL);
    {
      ScopedThreadStateChange tsc(self, kNative);
      ScopedLocalRef<jobject> class_name_object(soa.Env(),
                                              soa.Env()->NewStringUTF(class_name_string.c_str()));
      if (class_name_object.get() == NULL) {
        return NULL;
      }
      CHECK(class_loader_object.get() != NULL);
      result.reset(soa.Env()->CallObjectMethod(class_loader_object.get(),
                                             WellKnownClasses::java_lang_ClassLoader_loadClass,
                                             class_name_object.get()));
    }
    if (soa.Self()->IsExceptionPending()) {
      return NULL;
    } else if (result.get() == NULL) {
      ThrowNullPointerException(NULL, StringPrintf("ClassLoader.loadClass returned null for %s",
                                             class_name_string.c_str()).c_str());
      return NULL;
    } else {
      return soa.Decode<mirror::Class*>(result.get());
    }
  }

  ThrowNoClassDefFoundError("Class %s not found", PrintableString(descriptor).c_str());
  return NULL;
}
```

函数 LookupClassFromImage()也在文件 art/runtime/class_linker.cc 中定义，具体实现代码如下所示。

```
mirror::Class* ClassLinker::LookupClassFromImage(const char* descriptor) {
  Thread* self = Thread::Current();
  const char* old_no_suspend_cause =
      self->StartAssertNoThreadSuspension("Image class lookup");
  mirror::ObjectArray<mirror::DexCache>* dex_caches = GetImageDexCaches();
  for (int32_t i = 0; i < dex_caches->GetLength(); ++i) {
    mirror::DexCache* dex_cache = dex_caches->Get(i);
    const DexFile* dex_file = dex_cache->GetDexFile();
    if (descriptor[0] == 'L') {
      const DexFile::StringId* descriptor_string_id = dex_file->FindStringId(descriptor);
      if (descriptor_string_id != NULL) {
        const DexFile::TypeId* type_id =
            dex_file->FindTypeId(dex_file->GetIndexForStringId(*descriptor_string_id));
        if (type_id != NULL) {
```

```
            mirror::Class* klass = dex_cache->GetResolvedType(dex_file->GetIndexForTypeId(*type_id));
            if (klass != NULL) {
              self->EndAssertNoThreadSuspension(old_no_suspend_cause);
              return klass;
            }
          }
        }
      }
      const DexFile::StringId* string_id = dex_file->FindStringId(descriptor);
      if (string_id != NULL) {
        const DexFile::TypeId* type_id =
            dex_file->FindTypeId(dex_file->GetIndexForStringId(*string_id));
        if (type_id != NULL) {
          uint16_t type_idx = dex_file->GetIndexForTypeId(*type_id);
          mirror::Class* klass = dex_cache->GetResolvedType(type_idx);
          if (klass != NULL) {
            self->EndAssertNoThreadSuspension(old_no_suspend_cause);
            return klass;
          }
        }
      }
    }
    self->EndAssertNoThreadSuspension(old_no_suspend_cause);
    return NULL;
}
```

10.3.2　函数 DefineClass()

函数 DefineClass()在文件 art/runtime/class_linker.cc 中定义，实现 LoadClass()、InsertClass()和 LinkClass()等动作。其中，LoadClass()调用 LoadField()和 LoadMethod()等函数把类中的域和方法数据从 DEX 文件中读出来，填入 Class 结构。

函数 DefineClass()的具体实现代码如下所示。

```
mirror::Class* ClassLinker::DefineClass(const char* descriptor,
                                        mirror::ClassLoader* class_loader,
                                        const DexFile& dex_file,
                                        const DexFile::ClassDef& dex_class_def) {
  Thread* self = Thread::Current();
  SirtRef<mirror::Class> klass(self, NULL);
  if (UNLIKELY(!init_done_)) {
    if (strcmp(descriptor, "Ljava/lang/Object;") == 0) {
      klass.reset(GetClassRoot(kJavaLangObject));
    } else if (strcmp(descriptor, "Ljava/lang/Class;") == 0) {
      klass.reset(GetClassRoot(kJavaLangClass));
    } else if (strcmp(descriptor, "Ljava/lang/String;") == 0) {
      klass.reset(GetClassRoot(kJavaLangString));
    } else if (strcmp(descriptor, "Ljava/lang/DexCache;") == 0) {
```

```
            klass.reset(GetClassRoot(kJavaLangDexCache));
        } else if (strcmp(descriptor, "Ljava/lang/reflect/ArtField;") == 0) {
            klass.reset(GetClassRoot(kJavaLangReflectArtField));
        } else if (strcmp(descriptor, "Ljava/lang/reflect/ArtMethod;") == 0) {
            klass.reset(GetClassRoot(kJavaLangReflectArtMethod));
        } else {
            klass.reset(AllocClass(self, SizeOfClass(dex_file, dex_class_def)));
        }
    } else {
        klass.reset(AllocClass(self, SizeOfClass(dex_file, dex_class_def)));
    }
    if (UNLIKELY(klass.get() == NULL)) {
        CHECK(self->IsExceptionPending());   // Expect an OOME.
        return NULL;
    }
    klass->SetDexCache(FindDexCache(dex_file));
    LoadClass(dex_file, dex_class_def, klass, class_loader);
    if (self->IsExceptionPending()) {
        klass->SetStatus(mirror::Class::kStatusError, self);
        return NULL;
    }
    ObjectLock lock(self, klass.get());
    klass->SetClinitThreadId(self->GetTid());
    {
        mirror::Class* existing = InsertClass(descriptor, klass.get(), Hash(descriptor));
        if (existing != NULL) {
            return EnsureResolved(self, existing);
        }
    }
    CHECK(!klass->IsLoaded());
    if (!LoadSuperAndInterfaces(klass, dex_file)) {
        klass->SetStatus(mirror::Class::kStatusError, self);
        return NULL;
    }
    CHECK(klass->IsLoaded());
    CHECK(!klass->IsResolved());
    if (!LinkClass(klass, NULL, self)) {
        klass->SetStatus(mirror::Class::kStatusError, self);
        return NULL;
    }
    CHECK(klass->IsResolved());
    Dbg::PostClassPrepare(klass.get());

    return klass.get();
}
```

函数 LoadClass()也是在文件 art/runtime/class_linker.cc 中定义的，具体实现代码如下所示。

```
void ClassLinker::LoadClass(const DexFile& dex_file,
                            const DexFile::ClassDef& dex_class_def,
                            SirtRef<mirror::Class>& klass,
                            mirror::ClassLoader* class_loader) {
  CHECK(klass.get() != NULL);
  CHECK(klass->GetDexCache() != NULL);
  CHECK_EQ(mirror::Class::kStatusNotReady, klass->GetStatus());
  const char* descriptor = dex_file.GetClassDescriptor(dex_class_def);
  CHECK(descriptor != NULL);

  klass->SetClass(GetClassRoot(kJavaLangClass));
  uint32_t access_flags = dex_class_def.access_flags_;
  CHECK_EQ(access_flags & ~kAccJavaFlagsMask, 0U);
  klass->SetAccessFlags(access_flags);
  klass->SetClassLoader(class_loader);
  DCHECK_EQ(klass->GetPrimitiveType(), Primitive::kPrimNot);
  klass->SetStatus(mirror::Class::kStatusIdx, NULL);

  klass->SetDexClassDefIndex(dex_file.GetIndexForClassDef(dex_class_def));
  klass->SetDexTypeIndex(dex_class_def.class_idx_);

  const byte* class_data = dex_file.GetClassData(dex_class_def);
  if (class_data == NULL) {
    return;
  }
  ClassDataItemIterator it(dex_file, class_data);
  Thread* self = Thread::Current();
  if (it.NumStaticFields() != 0) {
    mirror::ObjectArray<mirror::ArtField>* statics = AllocArtFieldArray(self, it.NumStaticFields());
    if (UNLIKELY(statics == NULL)) {
      CHECK(self->IsExceptionPending());
      return;
    }
    klass->SetSFields(statics);
  }
  if (it.NumInstanceFields() != 0) {
    mirror::ObjectArray<mirror::ArtField>* fields =
        AllocArtFieldArray(self, it.NumInstanceFields());
    if (UNLIKELY(fields == NULL)) {
      CHECK(self->IsExceptionPending());  // OOME
      return;
    }
    klass->SetIFields(fields);
  }
  for (size_t i = 0; it.HasNextStaticField(); i++, it.Next()) {
    SirtRef<mirror::ArtField> sfield(self, AllocArtField(self));
    if (UNLIKELY(sfield.get() == NULL)) {
      CHECK(self->IsExceptionPending());  // OOME
      return;
    }
```

```
    klass->SetStaticField(i, sfield.get());
    LoadField(dex_file, it, klass, sfield);
  }
  for (size_t i = 0; it.HasNextInstanceField(); i++, it.Next()) {
    SirtRef<mirror::ArtField> ifield(self, AllocArtField(self));
    if (UNLIKELY(ifield.get() == NULL)) {
      CHECK(self->IsExceptionPending());   // OOME
      return;
    }
    klass->SetInstanceField(i, ifield.get());
    LoadField(dex_file, it, klass, ifield);
  }

  UniquePtr<const OatFile::OatClass> oat_class;
  if (Runtime::Current()->IsStarted() && !Runtime::Current()->UseCompileTimeClassPath()) {
    oat_class.reset(GetOatClass(dex_file, klass->GetDexClassDefIndex()));
  }

  if (it.NumDirectMethods() != 0) {
    mirror::ObjectArray<mirror::ArtMethod>* directs =
          AllocArtMethodArray(self, it.NumDirectMethods());
    if (UNLIKELY(directs == NULL)) {
      CHECK(self->IsExceptionPending());   // OOME
      return;
    }
    klass->SetDirectMethods(directs);
  }
  if (it.NumVirtualMethods() != 0) {
    // TODO: append direct methods to class object
    mirror::ObjectArray<mirror::ArtMethod>* virtuals =
         AllocArtMethodArray(self, it.NumVirtualMethods());
    if (UNLIKELY(virtuals == NULL)) {
      CHECK(self->IsExceptionPending());   // OOME
      return;
    }
    klass->SetVirtualMethods(virtuals);
  }
  size_t class_def_method_index = 0;
  for (size_t i = 0; it.HasNextDirectMethod(); i++, it.Next()) {
    SirtRef<mirror::ArtMethod> method(self, LoadMethod(self, dex_file, it, klass));
    if (UNLIKELY(method.get() == NULL)) {
      CHECK(self->IsExceptionPending());   // OOME
      return;
    }
    klass->SetDirectMethod(i, method.get());
    if (oat_class.get() != NULL) {
      LinkCode(method, oat_class.get(), class_def_method_index);
    }
    method->SetMethodIndex(class_def_method_index);
```

```
    class_def_method_index++;
  }
  for (size_t i = 0; it.HasNextVirtualMethod(); i++, it.Next()) {
    SirtRef<mirror::ArtMethod> method(self, LoadMethod(self, dex_file, it, klass));
    if (UNLIKELY(method.get() == NULL)) {
      CHECK(self->IsExceptionPending());   // OOME
      return;
    }
    klass->SetVirtualMethod(i, method.get());
    DCHECK_EQ(class_def_method_index, it.NumDirectMethods() + i);
    if (oat_class.get() != NULL) {
      LinkCode(method, oat_class.get(), class_def_method_index);
    }
    class_def_method_index++;
  }
  DCHECK(!it.HasNext());
}
```

10.3.3　函数 InsertClass()

函数 InsertClass()在文件 art/runtime/class_linker.cc 中定义，主要功能是把该类写入 class_table_中方便下次查找。函数 InsertClass()的具体实现代码如下所示。

```
mirror::Class* ClassLinker::InsertClass(const char* descriptor, mirror::Class* klass,
                                        size_t hash) {
  if (VLOG_IS_ON(class_linker)) {
    mirror::DexCache* dex_cache = klass->GetDexCache();
    std::string source;
    if (dex_cache != NULL) {
      source += " from ";
      source += dex_cache->GetLocation()->ToModifiedUtf8();
    }
    LOG(INFO) << "Loaded class " << descriptor << source;
  }
  WriterMutexLock mu(Thread::Current(), *Locks::classlinker_classes_lock_);
  mirror::Class* existing =
      LookupClassFromTableLocked(descriptor, klass->GetClassLoader(), hash);
  if (existing != NULL) {
    return existing;
  }
  if (kIsDebugBuild && klass->GetClassLoader() == NULL && dex_cache_image_class_lookup_required_) {
    existing = LookupClassFromImage(descriptor);
    if (existing != NULL) {
      CHECK(klass == existing);
    }
  }
  Runtime::Current()->GetHeap()->VerifyObject(klass);
  class_table_.insert(std::make_pair(hash, klass));
```

```
  class_table_dirty_ = true;
  return NULL;
}
```

10.3.4 函数 LinkClass()

函数 LinkClass()在文件 art/runtime/class_linke.cc 中定义，功能是动态绑定虚函数和接口函数，其调用结构如下所示。

```
LinkSuperClass() //检查父类
LinkMethods()
LinkVirtualMethods() //结合父类进行虚函数绑定，填写 Class 中的虚函数表 vtable_
LinkInterfaceMethods() //处理接口类函数信息 iftable_。注意接口类中的虚函数也会影响虚函数表，因此会更新
vtable_
LinkInstanceFields() & LinkStaticFields() //更新域信息，如域中的 Offset 和类的对象大小等
```

函数 LinkClass()的具体实现代码如下所示。

```
bool ClassLinker::LinkClass(SirtRef<mirror::Class>& klass,
                            mirror::ObjectArray<mirror::Class>* interfaces, Thread* self) {
  CHECK_EQ(mirror::Class::kStatusLoaded, klass->GetStatus());
  if (!LinkSuperClass(klass)) {
    return false;
  }
  if (!LinkMethods(klass, interfaces)) {
    return false;
  }
  if (!LinkInstanceFields(klass)) {
    return false;
  }
  if (!LinkStaticFields(klass)) {
    return false;
  }
  CreateReferenceInstanceOffsets(klass);
  CreateReferenceStaticOffsets(klass);
  CHECK_EQ(mirror::Class::kStatusLoaded, klass->GetStatus());
  klass->SetStatus(mirror::Class::kStatusResolved, self);
  return true;
}
```

对于函数 FindClass()来说，总共包含了内置类、启动类、系统类和其他类。其中，内置类是很基本的类，一般是初始化时预加载好的（如 WellKnownClasses 和 JniConstants 中的类），可以通过 LookupClassFromImage()函数找到。启动类是在 BOOTCLASSPATH 中的类，由于是启动类，所以这里还没有 ClassLoader。除掉前面的内置类，其余的通过 DexFile::FindInClassPath()查找得到。而系统类和其他类的加载过程是类似的，都是通过 ClassLoader 的 loadClass()方法加载，区别在于前者通过特殊的 SystemClassLoader 进行加载。例如，对于一个还没被加载过的启动类来说，一般流程如图 10-4 所示。

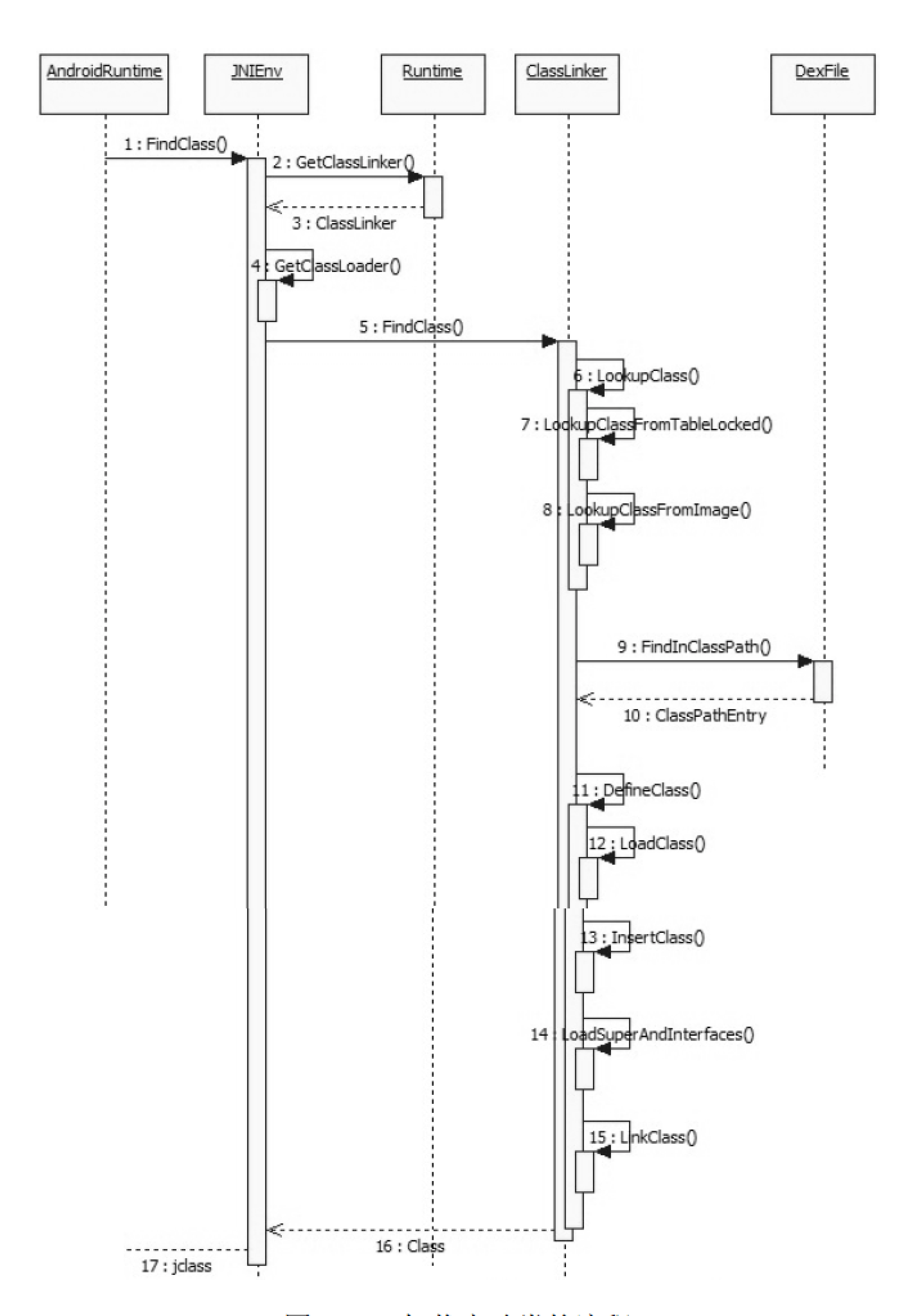


图 10-4　加载启动类的流程

整个过程涉及很多类，其中最主要的是 Class 类，具体结构如图 10-5 所示。

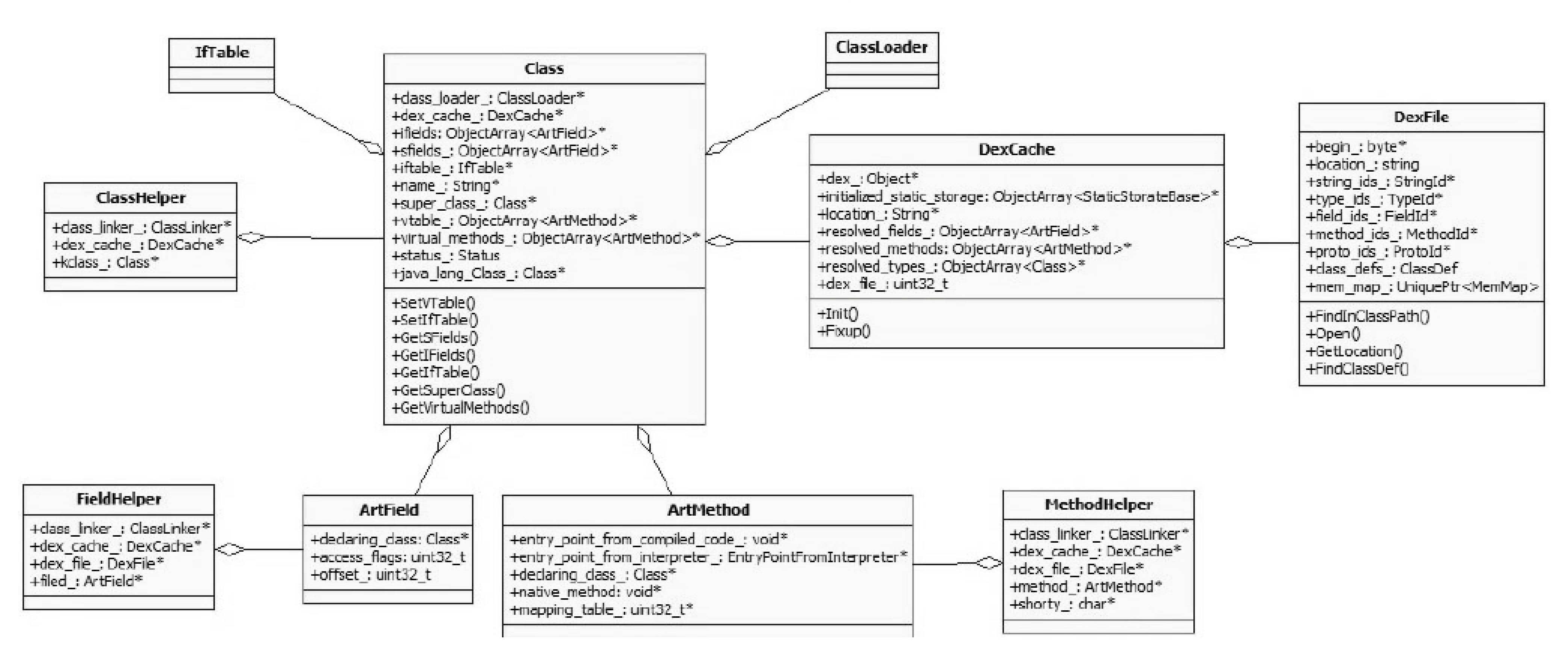


图 10-5　类结构关系

10.4 类 操 作

再次回到 FindClass()函数，因为在调用 ZygoteInit.main()时，所需的类在初始化时都已经装载好了，所以此处不按照上面的流程进行，而是直接通过 JNI 调用 ClassLoader.loadClass()进行装载。完成后将找到的类转为jclass 返回给 AndroidRuntime。类的查找工作结束后可以找相应的方法。GetStaticMethodID 会调用 FindMethodID()函数，此函数首先对该类进行验证，保证这个类是初始化好的，再调用其他函数进行目标函数的查找。

其中，函数 GetStaticMethodID()在文件 jni_internal.cc 中定义，能够将未初始化的类初始化，获取静态函数 main()的 ID。具体实现代码如下所示。

```
static jmethodID GetStaticMethodID(JNIEnv* env, jclass java_class, const char* name,
                                     const char* sig) {
    CHECK_NON_NULL_ARGUMENT(GetStaticMethodID, java_class);
    CHECK_NON_NULL_ARGUMENT(GetStaticMethodID, name);
    CHECK_NON_NULL_ARGUMENT(GetStaticMethodID, sig);
    ScopedObjectAccess soa(env);
    return FindMethodID(soa, java_class, name, sig, true);
}
```

FindMethodID()函数也在文件 jni_internal.cc 中定义，具体实现代码如下所示。

```
static jmethodID FindMethodID(ScopedObjectAccess& soa, jclass jni_class,
                              const char* name, const char* sig, bool is_static)
    SHARED_LOCKS_REQUIRED(Locks::mutator_lock_) {
  Class* c = soa.Decode<Class*>(jni_class);
  if (!Runtime::Current()->GetClassLinker()->EnsureInitialized(c, true, true)) {
    return NULL;
  }

  ArtMethod* method = NULL;
  if (is_static) {
    method = c->FindDirectMethod(name, sig);
  } else {
    method = c->FindVirtualMethod(name, sig);
    if (method == NULL) {
      method = c->FindDeclaredDirectMethod(name, sig);
    }
  }

  if (method == NULL || method->IsStatic() != is_static) {
    ThrowNoSuchMethodError(soa, c, name, sig, is_static ? "static" : "non-static");
    return NULL;
  }
```

```
  return soa.EncodeMethod(method);
}
```

通过上述实现代码可知，会根据不同的需求执行不同的函数，具体说明如下所示。

☑ 如果要找的是静态函数（通过 GetStaticMethodID()传递），则调用 FindDirectMethod()查找该类及其父类的非虚函数（通过 Class 的成员变量 direct_methods_）。

☑ 否则调用 FindVirtualMethod()查找该类及其父类的虚函数（通过 Class 的成员变量 virtual_methods_），如果没找到，再调用 FindDeclaredDirectMethod()查找该类的非虚函数。找到的条件是函数名和函数签名相同，例如，这里 main 和([Ljava/lang/String;)V。找到目标函数后即可执行。

函数 FindDirectMethod()的具体定义代码如下所示。

```
-Method* Class::FindDirectMethod(const StringPiece& name,
-                                 const StringPiece& signature) {
-  for (Class* klass = this; klass != NULL; klass = klass->GetSuperClass()) {
+Method* Class::FindDeclaredDirectMethod(const DexCache* dex_cache, uint32_t dex_method_idx) const {
+  if (GetDexCache() == dex_cache) {
+    for (size_t i = 0; i < NumDirectMethods(); ++i) {
+      Method* method = GetDirectMethod(i);
+      if (method->GetDexMethodIndex() == dex_method_idx) {
+        return method;
+      }
+    }
+  }
+  return NULL;
+}
```

函数 FindDeclaredDirectMethod()的具体定义代码如下所示。

```
-Method* Class::FindDeclaredDirectMethod(const StringPiece& name,
-                                         const StringPiece& signature) {
+Method* Class::FindInterfaceMethod(const DexCache* dex_cache, uint32_t dex_method_idx) const {
+  // Check the current class before checking the interfaces.
+  Method* method = FindDeclaredVirtualMethod(dex_cache, dex_method_idx);
+  if (method != NULL) {
+    return method;
+  }
+
+  int32_t iftable_count = GetIfTableCount();
+  ObjectArray<InterfaceEntry>* iftable = GetIfTable();
+  for (int32_t i = 0; i < iftable_count; i++) {
+    method = iftable->Get(i)->GetInterface()->FindVirtualMethod(dex_cache, dex_method_idx);
+    if (method != NULL) {
+      return method;
+    }
+  }
+  return NULL;
+}
```

10.5 实现托管操作

开始执行托管操作，函数 InvokeMain()会验证 Java 的 main()方法，并最终调用 CallStaticVoidMethod()来运行 main()方法。CallStaticVoidMethod(jni.h)在结构_JNIEnv 中实现，函数 CallStaticVoidMethod()在文件 jni_internal.cc 中定义，具体实现代码如下所示。

```
static void CallStaticVoidMethod(JNIEnv* env, jclass, jmethodID mid, ...) {
    va_list ap;
    va_start(ap, mid);
    CHECK_NON_NULL_ARGUMENT(CallStaticVoidMethod, mid);
    ScopedObjectAccess soa(env);
    InvokeWithVarArgs(soa, NULL, mid, ap);
    va_end(ap);
}
```

在上述代码中，va_list 用于处理不定传参数，function 表示 JNINativeInterface 结构指针表，用于保存 JNI 接口函数，例如调用 method 等。

函数 CallStaticVoidMethodV()在文件 JNI_interl.cc 中定义，具体实现代码如下所示。

```
static void CallStaticVoidMethodV(JNIEnv* env, jclass, jmethodID mid, va_list args) {
    CHECK_NON_NULL_ARGUMENT(CallStaticVoidMethodV, mid);
    ScopedObjectAccess soa(env);
    InvokeWithVarArgs(soa, NULL, mid, args);
}
```

函数 InvokeWithArgArray()也在文件 JNI_interl.cc 中定义，具体实现代码如下所示。

```
void InvokeWithArgArray(const ScopedObjectAccess& soa, ArtMethod* method,
                        ArgArray* arg_array, JValue* result, char result_type)
    SHARED_LOCKS_REQUIRED(Locks::mutator_lock_) {
  uint32_t* args = arg_array->GetArray();
  if (UNLIKELY(soa.Env()->check_jni)) {
    CheckMethodArguments(method, args);
  }
  method->Invoke(soa.Self(), args, arg_array->GetNumBytes(), result, result_type);
}
```

接下来执行文件 art/runtime/mirrorart_method.cc 中的函数 Invoke()，具体实现代码如下所示。

```
void ArtMethod::Invoke(Thread* self, uint32_t* args, uint32_t args_size, JValue* result,
                       char result_type) {
  if (kIsDebugBuild) {
    self->AssertThreadSuspensionIsAllowable(); //设定 debug 时线程可以被 hold
    CHECK_EQ(kRunnable, self->GetState());
  }

  ManagedStack fragment;
```

```
    self->PushManagedStackFragment(&fragment); //管理栈帧：this 放入 fragment,this 清空，保存现场

    Runtime* runtime = Runtime::Current();
    if (UNLIKELY(!runtime->IsStarted())) {
      LOG(INFO) << "Not invoking " << PrettyMethod(this) << " for a runtime that isn't started";
      if (result != NULL) {
        result->SetJ(0);
      }
    } else {
      const bool kLogInvocationStartAndReturn = false;
      if (GetEntryPointFromCompiledCode() != NULL) {        //存在被编译的 code
        if (kLogInvocationStartAndReturn) {
          LOG(INFO) << StringPrintf("Invoking '%s' code=%p", PrettyMethod(this).c_str(), GetEntryPointFrom
CompiledCode());
        }
#ifdef ART_USE_PORTABLE_COMPILER
        (*art_portable_invoke_stub)(this, args, args_size, self, result, result_type);
#else
        (*art_quick_invoke_stub)(this, args, args_size, self, result, result_type);
#endif
        if (UNLIKELY(reinterpret_cast<int32_t>(self->GetException(NULL)) == -1)) {
          //在生成 llvm 代码过程中如果异常会进入解释器
          self->ClearException();
          ShadowFrame* shadow_frame = self->GetAndClearDeoptimizationShadowFrame(result);
          self->SetTopOfStack(NULL, 0);
          self->SetTopOfShadowStack(shadow_frame);                              //stack & shadow frame 设置
          interpreter::EnterInterpreterFromDeoptimize(self, shadow_frame, result);      //在解释器继续执行
        }
        if (kLogInvocationStartAndReturn) {
          LOG(INFO) << StringPrintf("Returned '%s' code=%p", PrettyMethod(this).c_str(), GetEntryPointFrom
CompiledCode());
        }
      } else {
        LOG(INFO) << "Not invoking '" << PrettyMethod(this)
            << "' code=" << reinterpret_cast<const void*>(GetEntryPointFromCompiledCode());
        if (result != NULL) {
          result->SetJ(0);
        }
      }
    }

    // Pop transition.
    self->PopManagedStackFragment(fragment); //恢复现场
}
```

在上述代码中，前后分别实现了对托管代码栈的保存和恢复工作。

接下来进入解释器的函数 EnterInterpreterFromDeoptimize()，具体实现代码如下所示。

```
void EnterInterpreterFromDeoptimize(Thread* self, ShadowFrame* shadow_frame, JValue* ret_val)
    SHARED_LOCKS_REQUIRED(Locks::mutator_lock_) {
  JValue value;
  value.SetJ(ret_val->GetJ());
  MethodHelper mh;                                          //method 操作的 class
  while (shadow_frame != NULL) {
    self->SetTopOfShadowStack(shadow_frame);
    mh.ChangeMethod(shadow_frame->GetMethod());             //获取 method
    const DexFile::CodeItem* code_item = mh.GetCodeItem();  //code 传递
    value = Execute(self, mh, code_item, *shadow_frame, value);  //执行解释器
    ShadowFrame* old_frame = shadow_frame;
    shadow_frame = shadow_frame->GetLink();                 //下一个 frame 赋值给 shadow_frame
    delete old_frame;                                       //删除前一个 frame
  }
  ret_val->SetJ(value.GetJ());
}
```

函数 Execute()在文件 interpretor.cc 中定义，具体实现代码如下所示。

```
static inline JValue Execute(Thread* self, MethodHelper& mh, const DexFile::CodeItem* code_item,
                             ShadowFrame& shadow_frame, JValue result_register) {
  DCHECK(shadow_frame.GetMethod() == mh.GetMethod() ||
         shadow_frame.GetMethod()->GetDeclaringClass()->IsProxyClass());
  DCHECK(!shadow_frame.GetMethod()->IsAbstract());
  DCHECK(!shadow_frame.GetMethod()->IsNative());
  if (shadow_frame.GetMethod()->IsPreverified()) {          //是否提前做过 method access verify
    return ExecuteImpl<false>(self, mh, code_item, shadow_frame, result_register);  //进入具体实现函数，可
以发现是一个 C 语言解释器
  } else {
    return ExecuteImpl<true>(self, mh, code_item, shadow_frame, result_register);  //进入具体实现函数，是一
个 C 语言解释器
  }
}
```

再回到前面文件 art/runtime/mirrorart_method.cc 中的函数 invoke()，通过 ArtMethod 类的成员函数 Invoke 来调用参数 method 指定的类方法。具体代码如下所示。

```
const bool kLogInvocationStartAndReturn = false;
if (GetEntryPointFromCompiledCode() != NULL) {           //存在被编译的code
    if (kLogInvocationStartAndReturn) {
      LOG(INFO) << StringPrintf("Invoking '%s' code=%p", PrettyMethod(this).c_str(), GetEntryPointFrom
CompiledCode());
    }
#ifdef ART_USE_PORTABLE_COMPILER                         //portable 编译器
    (*art_portable_invoke_stub)(this, args, args_size, self, result, result_type);
#else
    (*art_quick_invoke_stub)(this, args, args_size, self, result, result_type);
#endif
    if (UNLIKELY(reinterpret_cast<int32_t>(self->GetException(NULL)) == -1)) {
    //生成 llvm 代码过程中产生异常会进入解释器
```

上述两个分支分别代表函数 art_portable_invoke_stub()和 art_quick_invoke_stub()，ART_USE_



PORTABLE_COMPILER 是一个重要的宏。

在执行托管代码前，要先为其创建栈。这些栈通过 ManagedStack 的成员 link_形成一个先入后出的链表。当执行完托管代码后，只要将最近放入的托管代码栈恢复即可。中间是目标函数的执行，但在跳入目标函数体前还需要先执行一些 ABI 层的上下文处理代码，这段代码称为 stub。首先按 ART_USE_PORTABLE_COMPILER 来决定是用 art_quick_invok_stub 还是 art_portable_invok_stub。由于是由汇编语言写成，平台相关，所以每个体系结构(x86, arm, mps)都有其实现。以 x86 体系为例，art_portable_invoke_stub 定义在文件 portable_entrypoints_x86.S 中，具体实现代码如下所示。

```
DEFINE_FUNCTION art_portable_invoke_stub
    PUSH ebp
    PUSH ebx
    mov %esp, %ebp
    .cfi_def_cfa_register ebp
    mov 20(%ebp), %ebx
    addl LITERAL(28), %ebx
    andl LITERAL(0xFFFFFFF0), %ebx
    subl LITERAL(12), %ebx
    subl %ebx, %esp
    lea   4(%esp), %eax
    pushl 20(%ebp)
    pushl 16(%ebp)
    pushl %eax
    call SYMBOL(memcpy)
    addl LITERAL(12), %esp
    mov 12(%ebp), %eax
    mov %eax, (%esp)
    call *METHOD_CODE_OFFSET(%eax)
    mov %ebp, %esp
    POP ebx
    POP ebp
    mov 20(%esp), %ecx
    cmpl LITERAL(68), 24(%esp)
    je return_double_portable
    cmpl LITERAL(70), 24(%esp)
    je return_float_portable
    mov %eax, (%ecx)
    mov %edx, 4(%ecx)
    ret
return_double_portable:
    fstpl (%ecx)
    ret
return_float_portable:
    fstps (%ecx)
    ret
END_FUNCTION art_portable_invoke_stub
```

由此可见，这是 x86 体系中的函数调用过程。首先保存栈帧等信息，然后把参数数组复制到栈中，再执行 call 指令跳转到要执行的目标函数。METHOD_CODE_OFFSET 指向 ArtMethod 中的成员变量

entry_point_from_compiled_code_，也就是编译好的目标函数的地址。接下来就是等目标函数执行完，然后恢复上下文，保存返回值，最后执行 ret 指令返回。查找目标函数和执行的过程比较直观，如图 10-6 所示。

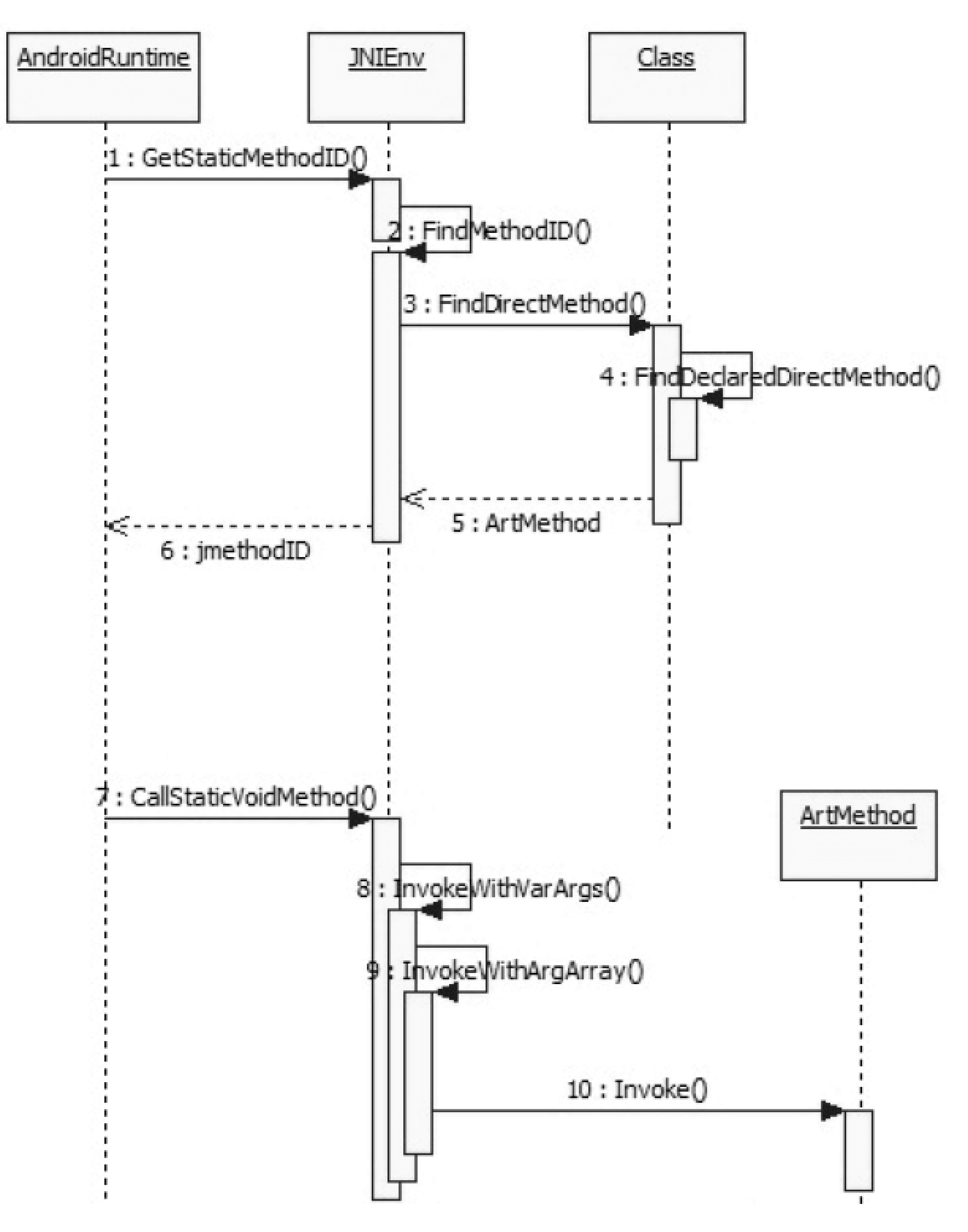


图 10-6　查找目标函数和执行的过程

到此为止，执行完托管代码后返回到 AndroidRuntimie::start()函数，调用函数 DetachCurrentThread()和 DestroyJavaVM()来做清理工作，并关闭虚拟机，完成整个工作过程。

```
ALOGD("Shutting down VM\n");
if (mJavaVM->DetachCurrentThread() != JNI_OK)
      ALOGW("Warning: unable to detach main thread\n");
if (mJavaVM->DestroyJavaVM() != 0)
      ALOGW("Warning: VM did not shut down cleanly\n");
```

函数 DetachCurrentThread()和 DestroyJavaVM()在文件 jni_internal.cc 中定义，具体实现代码如下所示。

```
static jint DetachCurrentThread(JavaVM* vm) {
      if (vm == NULL || Thread::Current() == NULL) {
         return JNI_ERR;
      }
      JavaVMExt* raw_vm = reinterpret_cast<JavaVMExt*>(vm);
      Runtime* runtime = raw_vm->runtime;
      runtime->DetachCurrentThread();
```

```
    return JNI_OK;
  }
public:
  static jint DestroyJavaVM(JavaVM* vm) {
    if (vm == NULL) {
      return JNI_ERR;
    }
    JavaVMExt* raw_vm = reinterpret_cast<JavaVMExt*>(vm);
    delete raw_vm->runtime;
    return JNI_OK;
}
```

在 Android 系统中，当在一个线程中调用 AttachCurrentThread 后，如果不需要用时一定要做 DetachCurrentThread 处理，否则线程无法正常退出。

第 11 章　Sensor 传感器系统架构详解

传感器是近年来随着物联网这一概念的流行而推出的，已经逐渐为人们所接受。其实传感器在日常的生活中经常见到甚至是用到，例如楼宇的声控楼梯灯和马路上的路灯等。Android 5.0 中的传感器系统是 Sensor，在 Android 系统中提供的传感器主要有加速度、磁场、方向、陀螺仪、光线、压力、温度和距离传感器等。

本章将详细讲解 Android 5.0 系统中传感器系统的基本知识，为读者学习本书后面的知识打下基础。

11.1　Android 传感器系统概述

传感器系统会主动对上层报告传感器精度和数据的变化，并且提供了设置传感器精度的接口，这些接口可以在 Java 应用和 Java 框架中使用。Android 传感器系统的基本层次结构如图 11-1 所示。

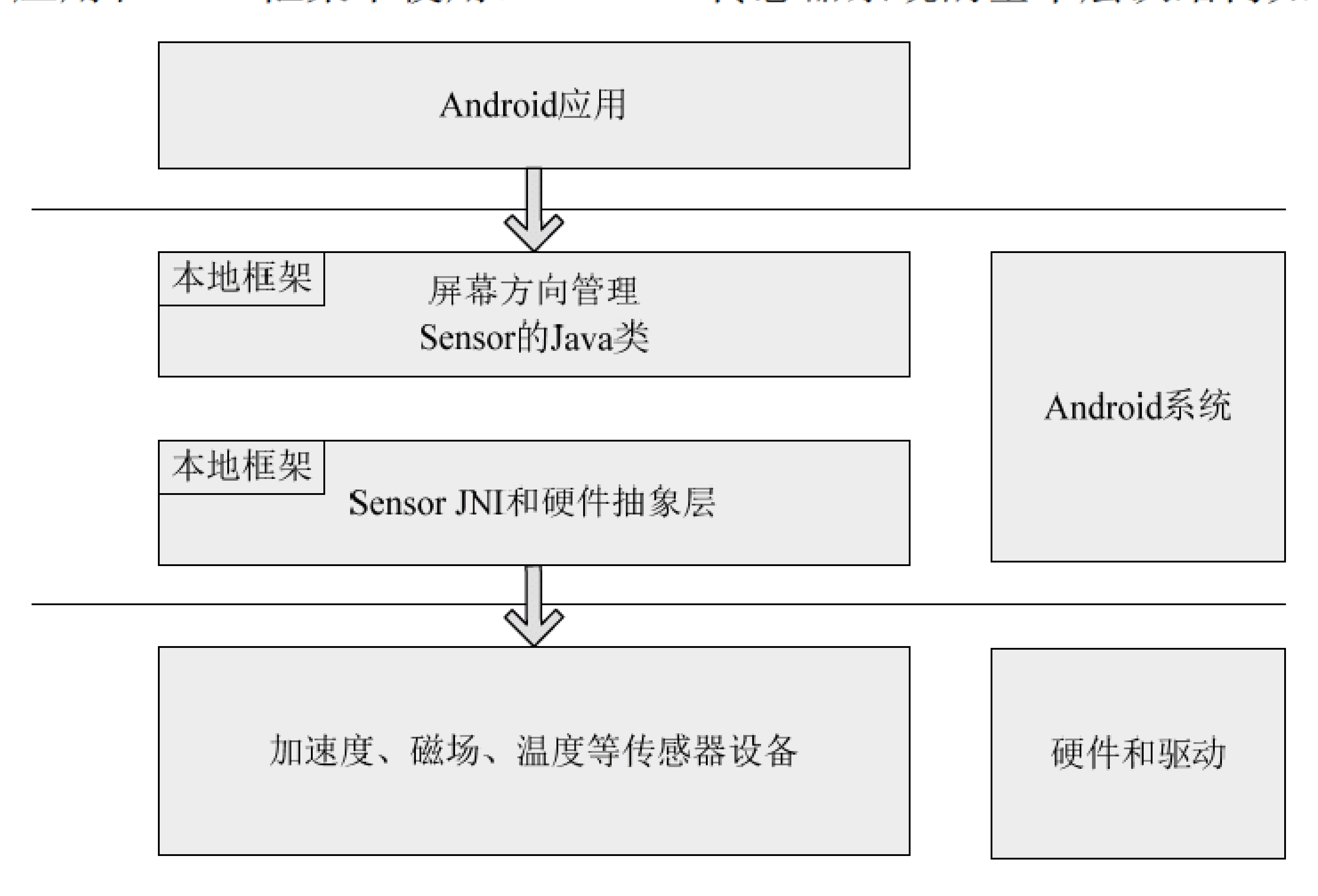


图 11-1　传感器系统的层次结构

根据图 11-1 所示的结构，Android 传感器系统从上到下分别是 Java 应用层、Java 框架对传感器的应用、传感器类、传感器硬件抽象层、传感器驱动。各个层的具体说明如下所示。

（1）传感器系统的 Java 部分

代码路径是 frameworks/base/include/core/java/android/hardware。

此部分对应的实现文件是 Sensor*.java。

（2）传感器系统的 JNI 部分

代码路径是 frameworks/base/core/jni/android_hardware_SensorManager.cpp。

在此部分中提供了对类 android.hardware.Sensor.Manage 的本地支持。

（3）传感器系统 HAL 层

头文件路径是 hardware/libhardware/include/hardware/sensors.h。

在 Android 系统中，传感器系统的硬件抽象层需要特意编码实现。

（4）驱动层

驱动层的代码路径是 kernel/driver/hwmon/$(PROJECT)/sensor。

在库 sensor.so 中提供了如下 8 个 API 函数。

☑ 控制方面：在结构体 ensors_control_device_t 中定义，包括如下函数。
  - int (*open_data_source)(struct sensors_control_device_t *dev)
  - int (*activate)(struct sensors_control_device_t *dev, int handle, int enabled)
  - int (*set_delay)(struct sensors_control_device_t *dev, int32_t ms)
  - int (*wake)(struct sensors_control_device_t *dev)

☑ 数据方面：在结构体 sensors_data_device_t 中定义，包括如下函数。
  - int (*data_open)(struct sensors_data_device_t *dev, int fd)
  - int (*data_close)(struct sensors_data_device_t *dev)
  - int (*poll)(struct sensors_data_device_t *dev, sensors_data_t* data)

☑ 模块方面：在结构体 sensors_module_t 中定义，包括如下函数。
  - int (*get_sensors_list)(struct sensors_module_t* module, struct sensor_t const** list)

在 Android 系统的 Java 层中，Sensor 的状态是由 SensorService 负责控制的，其 Java 代码和 JNI 代码分别位于文件 frameworks/base/services/java/com/android/server/SensorService.java 和 frameworks/base/services/jni/com_android_server_SensorService.cpp 中。

SensorManager 负责在 Java 层 Sensor 的数据控制，其 Java 代码和 JNI 代码分别位于文件 frameworks/base/core/java/android/hardware/SensorManager.java 和 frameworks/base/core/jni/android_hardware_SensorManager.cpp 中。

在 Android 的 Framework 中，是通过文件 sensorService.java 和 sensorManager.java 实现与 Sensor 传感器通信的。文件 sensorService.java 的通信功能是通过 JNI 调用 sensorService.cpp 中的方法实现的。

文件 sensorManager.java 的具体通信功能是通过 JNI 调用 sensorManager.cpp 中的方法实现的。文件 sensorService.cpp 和 sensorManager.cpp 通过文件 hardware.c 与 sensor.so 通信。其中，文件 sensorService.cpp 实现对 Sensor 的状态控制，文件 sensorManager.cpp 实现对 Sensor 的数据控制。

库 sensor.so 通过 ioctl 控制 sensor driver 的状态，通过打开 sensor driver 对应的设备文件读取 G-sensor 采集的数据。

11.2 Java 层详解

在 Android 系统中，传感器系统的 Java 部分的实现文件是/sdk/apps/SdkController/src/com/android/tools/sdkcontroller/activities/SensorActivity.java。

通过阅读文件 SensorActivity.java 的源码可知，在应用程序中使用传感器需要用到 hardware 包中的 SensorManager、SensorListener 等相关的类，具体实现代码如下所示。

```
public class SensorActivity extends BaseBindingActivity
        implements android.os.Handler.Callback {

    @SuppressWarnings("hiding")
    public static String TAG = SensorActivity.class.getSimpleName();
    private static boolean DEBUG = true;

    private static final int MSG_UPDATE_ACTUAL_HZ = 0x31415;

    private TableLayout mTableLayout;
    private TextView mTextError;
    private TextView mTextStatus;
    private TextView mTextTargetHz;
    private TextView mTextActualHz;
    private SensorChannel mSensorHandler;

    private final Map<MonitoredSensor, DisplayInfo> mDisplayedSensors =
        new HashMap<SensorChannel.MonitoredSensor, SensorActivity.DisplayInfo>();
    private final android.os.Handler mUiHandler = new android.os.Handler(this);
    private int mTargetSampleRate;
    private long mLastActualUpdateMs;

    /** 第一次创建 activity 时调用 */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.sensors);
        mTableLayout = (TableLayout) findViewById(R.id.tableLayout);
        mTextError    = (TextView) findViewById(R.id.textError);
        mTextStatus = (TextView) findViewById(R.id.textStatus);
        mTextTargetHz = (TextView) findViewById(R.id.textSampleRate);
        mTextActualHz = (TextView) findViewById(R.id.textActualRate);
        updateStatus("Waiting for connection");

        mTextTargetHz.setOnKeyListener(new OnKeyListener() {
            @Override
            public boolean onKey(View v, int keyCode, KeyEvent event) {
                updateSampleRate();
                return false;
            }
        });
        mTextTargetHz.setOnFocusChangeListener(new OnFocusChangeListener() {
            @Override
            public void onFocusChange(View v, boolean hasFocus) {
                updateSampleRate();
            }
        });
    }

    @Override
    protected void onResume() {
```

```
        if (DEBUG) Log.d(TAG, "onResume");
        //BaseBindingActivity 绑定后套服务
        super.onResume();
        updateError();
    }

    @Override
    protected void onPause() {
        if (DEBUG) Log.d(TAG, "onPause");
        super.onPause();
    }

    @Override
    protected void onDestroy() {
        if (DEBUG) Log.d(TAG, "onDestroy");
        super.onDestroy();
        removeSensorUi();
    }

    // ----------

    @Override
    protected void onServiceConnected() {
        if (DEBUG) Log.d(TAG, "onServiceConnected");
        createSensorUi();
    }

    @Override
    protected void onServiceDisconnected() {
        if (DEBUG) Log.d(TAG, "onServiceDisconnected");
        removeSensorUi();
    }

    @Override
    protected ControllerListener createControllerListener() {
        return new SensorsControllerListener();
    }

    // ----------

    private class SensorsControllerListener implements ControllerListener {
        @Override
        public void onErrorChanged() {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    updateError();
                }
            });
        }
```

```
        @Override
        public void onStatusChanged() {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    ControllerBinder binder = getServiceBinder();
                    if (binder != null) {
                        boolean connected = binder.isEmuConnected();
                        mTableLayout.setEnabled(connected);
                        updateStatus(connected ? "Emulated connected" : "Emulator disconnected");
                    }
                }
            });
        }
    }

    private void createSensorUi() {
        final LayoutInflater inflater = getLayoutInflater();

        if (!mDisplayedSensors.isEmpty()) {
            removeSensorUi();
        }

        mSensorHandler = (SensorChannel) getServiceBinder().getChannel(Channel.SENSOR_CHANNEL);
        if (mSensorHandler != null) {
            mSensorHandler.addUiHandler(mUiHandler);
            mUiHandler.sendEmptyMessage(MSG_UPDATE_ACTUAL_HZ);

            assert mDisplayedSensors.isEmpty();
            List<MonitoredSensor> sensors = mSensorHandler.getSensors();
            for (MonitoredSensor sensor : sensors) {
                final TableRow row = (TableRow) inflater.inflate(R.layout.sensor_row,
                                                                  mTableLayout,
                                                                  false);
                mTableLayout.addView(row);
                mDisplayedSensors.put(sensor, new DisplayInfo(sensor, row));
            }
        }
    }

    private void removeSensorUi() {
        if (mSensorHandler != null) {
            mSensorHandler.removeUiHandler(mUiHandler);
            mSensorHandler = null;
        }
        mTableLayout.removeAllViews();
        for (DisplayInfo info : mDisplayedSensors.values()) {
            info.release();
        }
        mDisplayedSensors.clear();
```

```
}

private class DisplayInfo implements CompoundButton.OnCheckedChangeListener {
    private MonitoredSensor mSensor;
    private CheckBox mChk;
    private TextView mVal;

    public DisplayInfo(MonitoredSensor sensor, TableRow row) {
        mSensor = sensor;

        mChk = (CheckBox) row.findViewById(R.id.row_checkbox);
        mChk.setText(sensor.getUiName());
        mChk.setEnabled(sensor.isEnabledByEmulator());
        mChk.setChecked(sensor.isEnabledByUser());
        mChk.setOnCheckedChangeListener(this);

        //初始化显示该传感器的文本框
        mVal = (TextView) row.findViewById(R.id.row_textview);
        mVal.setText(sensor.getValue());
    }

    /**
     *为相关的复选框选中状态进行变化的处理。当复选框被选中时会注册传感器变化
     *如果不加以控制会取消传感器的变化
     */
    @Override
    public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
        if (mSensor != null) {
            mSensor.onCheckedChanged(isChecked);
        }
    }

    public void release() {
        mChk = null;
        mVal = null;
        mSensor = null;

    }

    public void updateState() {
        if (mChk != null && mSensor != null) {
            mChk.setEnabled(mSensor.isEnabledByEmulator());
            mChk.setChecked(mSensor.isEnabledByUser());
        }
    }

    public void updateValue() {
        if (mVal != null && mSensor != null) {
            mVal.setText(mSensor.getValue());
        }
    }
```

```
}

/**实现回调处理程序*/
@Override
public boolean handleMessage(Message msg) {
    DisplayInfo info = null;
    switch (msg.what) {
    case SensorChannel.SENSOR_STATE_CHANGED:
        info = mDisplayedSensors.get(msg.obj);
        if (info != null) {
            info.updateState();
        }
        break;
    case SensorChannel.SENSOR_DISPLAY_MODIFIED:
        info = mDisplayedSensors.get(msg.obj);
        if (info != null) {
            info.updateValue();
        }
        if (mSensorHandler != null) {
            updateStatus(Integer.toString(mSensorHandler.getMsgSentCount()) + " events sent");

            //如果值已经修改，则更新 actual rate
            long ms = mSensorHandler.getActualUpdateMs();
            if (ms != mLastActualUpdateMs) {
                mLastActualUpdateMs = ms;
                String hz = mLastActualUpdateMs <= 0 ? "--" :
                                Integer.toString((int) Math.ceil(1000. / ms));
                mTextActualHz.setText(hz);
            }
        }
        break;
    case MSG_UPDATE_ACTUAL_HZ:
        if (mSensorHandler != null) {
            //如果值已经修改，则更新 actual rate
            long ms = mSensorHandler.getActualUpdateMs();
            if (ms != mLastActualUpdateMs) {
                mLastActualUpdateMs = ms;
                String hz = mLastActualUpdateMs <= 0 ? "--" :
                                Integer.toString((int) Math.ceil(1000. / ms));
                mTextActualHz.setText(hz);
            }
            mUiHandler.sendEmptyMessageDelayed(MSG_UPDATE_ACTUAL_HZ, 1000 /*1s*/);
        }
    }
    return true;
}

private void updateStatus(String status) {
    mTextStatus.setVisibility(status == null ? View.GONE : View.VISIBLE);
    if (status != null) mTextStatus.setText(status);
}
```

```
    private void updateError() {
        ControllerBinder binder = getServiceBinder();
        String error = binder == null ? "" : binder.getServiceError();
        if (error == null) {
            error = "";
        }

        mTextError.setVisibility(error.length() == 0 ? View.GONE : View.VISIBLE);
        mTextError.setText(error);
    }

    private void updateSampleRate() {
        String str = mTextTargetHz.getText().toString();
        try {
            int hz = Integer.parseInt(str.trim());

            if (hz <= 0 || hz > 50) {
                hz = 50;
            }

            if (hz != mTargetSampleRate) {
                mTargetSampleRate = hz;
                if (mSensorHandler != null) {
                    mSensorHandler.setUpdateTargetMs(hz <= 0 ? 0 : (int)(1000.0f / hz));
                }
            }
        } catch (Exception ignore) {}
    }
}
```

通过上述代码可知，整个 Java 层利用了观察者模式对传感器的数据进行了监听处理。

11.3　Frameworks 层详解

在 Android 系统中，传感器系统的 Frameworks 层的代码路径是 frameworks/base/include/core/java/android/hardware。

Frameworks 层是 Android 系统提供的应用程序开发接口和应用程序框架，与应用程序的调用是通过类实例化或类继承进行的。对应用程序来说，最重要的就是把 SensorListener 注册到 SensorManager 上，从而才能以观察者身份接收到数据的变化，因此，把注意力放在 SensorManager 的构造函数、RegisterListener()函数和通知机制相关的代码上。

本节将详细讲解传感器系统中 Frameworks 层的核心架构知识。

11.3.1　监听传感器的变化

在 Android 传感器系统的 Frameworks 层中，文件 SensorListener.java 用于监听从 Java 应用层中传

递过来的变化。文件 SensorListener.java 比较简单，具体代码如下所示。

```
package android.hardware;
@Deprecated
public interface SensorListener {
    public void onSensorChanged(int sensor, float[] values);
    public void onAccuracyChanged(int sensor, int accuracy);
}
```

11.3.2 注册监听

当文件 SensorListener.java 监听到变化之后，会通过文件 SensorManager.java 来向服务注册监听变化，并调度 Sensor 的具体任务。例如，在开发 Android 传感器应用程序时，上层的通用开发流程如下所示。

（1）通过 getSystemService(SENSOR_SERVICE);语句得到传感器服务，这样得到一个用来管理分配调度处理 Sensor 工作的 SensorManager。SensorManager 并不服务运行于后台，真正属于 Sensor 的系统服务是 SensorService，在终端下的#service list 中可以看到 sensorservice: [android.gui.SensorServer]。

（2）通过 getDefaultSensor(Sensor.TYPE_GRAVITY);得到传感器类型，当然还有各种千奇百怪的传感器，具体可以查阅 Android 官网 API 或者源码 Sensor.java。

（3）注册监听器 SensorEventListener。在应用程序中打开一个监听接口，专门用于处理传感器的数据。

（4）通过回调函数 onSensorChanged()和 onAccuracyChanged()实现实时监听，例如，对重力感应器的 x、y、z 值经算法变换得到左右、上下、前后方向等，就由这个回调函数实现。

综上所述，传感器顶层的处理流程如图 11-2 所示。

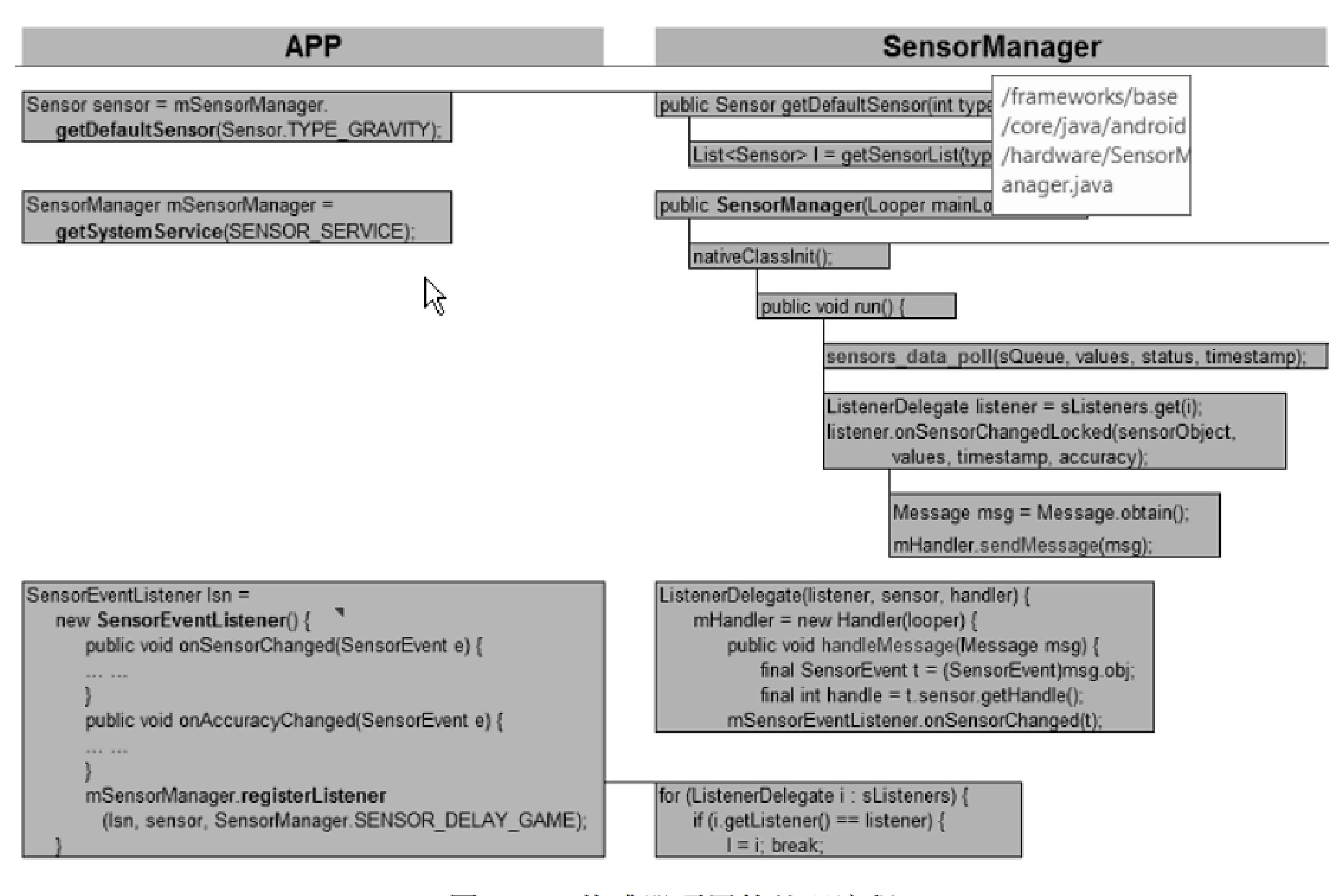
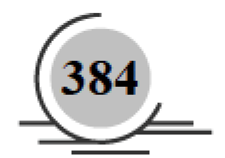


图 11-2 传感器顶层的处理流程



文件 SensorManager.java 的具体实现流程如下所示。

（1）定义类 SensorManager，然后设置各种传感器的初始变量值，具体代码如下所示。

```
public abstract class SensorManager {
    protected static final String TAG = "SensorManager";
    private static final float[] mTempMatrix = new float[16];

    private final SparseArray<List<Sensor>> mSensorListByType =
            new SparseArray<List<Sensor>>();

    private LegacySensorManager mLegacySensorManager;
    @Deprecated
    public static final int SENSOR_ORIENTATION = 1 << 0;
    @Deprecated
    public static final int SENSOR_ACCELEROMETER = 1 << 1;
    @Deprecated
    public static final int SENSOR_TEMPERATURE = 1 << 2;
    @Deprecated
    public static final int SENSOR_MAGNETIC_FIELD = 1 << 3;
    @Deprecated
    public static final int SENSOR_LIGHT = 1 << 4;
    @Deprecated
    public static final int SENSOR_PROXIMITY = 1 << 5;
    @Deprecated
    public static final int SENSOR_TRICORDER = 1 << 6;
    @Deprecated
    public static final int SENSOR_ORIENTATION_RAW = 1 << 7;
    @Deprecated
    public static final int SENSOR_ALL = 0x7F;
    @Deprecated
    public static final int SENSOR_MIN = SENSOR_ORIENTATION;
    @Deprecated
    public static final int SENSOR_MAX = ((SENSOR_ALL + 1)>>1);

    @Deprecated
    public static final int DATA_X = 0;
    @Deprecated
    public static final int DATA_Y = 1;
    @Deprecated
    public static final int DATA_Z = 2;
    @Deprecated
    public static final int RAW_DATA_INDEX = 3;
    @Deprecated
    public static final int RAW_DATA_X = 3;
    @Deprecated
    public static final int RAW_DATA_Y = 4;
    @Deprecated
    public static final int RAW_DATA_Z = 5;

    /** Standard gravity (g) on Earth. This value is equivalent to 1G */
```

```
public static final float STANDARD_GRAVITY = 9.80665f;

/** Sun's gravity in SI units (m/s^2) */
public static final float GRAVITY_SUN = 2712.0f;
/** Mercury's gravity in SI units (m/s^2) */
public static final float GRAVITY_MERCURY = 3.70f;
/** Venus' gravity in SI units (m/s^2) */
public static final float GRAVITY_VENUS = 8.87f;
/** Earth's gravity in SI units (m/s^2) */
public static final float GRAVITY_EARTH = 9.80665f;
/** The Moon's gravity in SI units (m/s^2) */
public static final float GRAVITY_MOON = 1.6f;
/** Mars' gravity in SI units (m/s^2) */
public static final float GRAVITY_MARS = 3.71f;
/** Jupiter's gravity in SI units (m/s^2) */
public static final float GRAVITY_JUPITER = 23.12f;
/** Saturn's gravity in SI units (m/s^2) */
public static final float GRAVITY_SATURN = 8.96f;
/** Uranus' gravity in SI units (m/s^2) */
public static final float GRAVITY_URANUS = 8.69f;
/** Neptune's gravity in SI units (m/s^2) */
public static final float GRAVITY_NEPTUNE = 11.0f;
/** Pluto's gravity in SI units (m/s^2) */
public static final float GRAVITY_PLUTO = 0.6f;
/** Gravity (estimate) on the first Death Star in Empire units (m/s^2) */
public static final float GRAVITY_DEATH_STAR_I = 0.000000353036145f;
/** Gravity on the island */
public static final float GRAVITY_THE_ISLAND = 4.815162342f;


/** Maximum magnetic field on Earth's surface */
public static final float MAGNETIC_FIELD_EARTH_MAX = 60.0f;
/** Minimum magnetic field on Earth's surface */
public static final float MAGNETIC_FIELD_EARTH_MIN = 30.0f;


/** Standard atmosphere, or average sea-level pressure in hPa (millibar) */
public static final float PRESSURE_STANDARD_ATMOSPHERE = 1013.25f;


/** Maximum luminance of sunlight in lux */
public static final float LIGHT_SUNLIGHT_MAX = 120000.0f;
/** luminance of sunlight in lux */
public static final float LIGHT_SUNLIGHT = 110000.0f;
/** luminance in shade in lux */
public static final float LIGHT_SHADE = 20000.0f;
/** luminance under an overcast sky in lux */
public static final float LIGHT_OVERCAST = 10000.0f;
/** luminance at sunrise in lux */
public static final float LIGHT_SUNRISE = 400.0f;
/** luminance under a cloudy sky in lux */
```

```
public static final float LIGHT_CLOUDY = 100.0f;
/** luminance at night with full moon in lux */
public static final float LIGHT_FULLMOON = 0.25f;
/** luminance at night with no moon in lux*/
public static final float LIGHT_NO_MOON = 0.001f;


/** get sensor data as fast as possible */
public static final int SENSOR_DELAY_FASTEST = 0;
/** rate suitable for games */
public static final int SENSOR_DELAY_GAME = 1;
/** rate suitable for the user interface   */
public static final int SENSOR_DELAY_UI = 2;
/**（默认值）适合屏幕方向的变化*/
public static final int SENSOR_DELAY_NORMAL = 3;


/**
 *返回的值，该传感器是不可信的，需要进行校准或环境不允许读数
 */
public static final int SENSOR_STATUS_UNRELIABLE = 0;

/**
 *该传感器是报告的低精度的数据，与环境的校准是必要的
 */
public static final int SENSOR_STATUS_ACCURACY_LOW = 1;

/**
 * This sensor is reporting data with an average level of accuracy,
 * calibration with the environment may improve the readings
 */
public static final int SENSOR_STATUS_ACCURACY_MEDIUM = 2;

/** This sensor is reporting data with maximum accuracy */
public static final int SENSOR_STATUS_ACCURACY_HIGH = 3;

/** see {@link #remapCoordinateSystem} */
public static final int AXIS_X = 1;
/** see {@link #remapCoordinateSystem} */
public static final int AXIS_Y = 2;
/** see {@link #remapCoordinateSystem} */
public static final int AXIS_Z = 3;
/** see {@link #remapCoordinateSystem} */
public static final int AXIS_MINUS_X = AXIS_X | 0x80;
/** see {@link #remapCoordinateSystem} */
public static final int AXIS_MINUS_Y = AXIS_Y | 0x80;
/** see {@link #remapCoordinateSystem} */
public static final int AXIS_MINUS_Z = AXIS_Z | 0x80;
```

（2）定义各种设备类型方法和设备数据的方法，这些方法非常重要，在编写的应用程序中，可以

通过 AIDL 接口远程调用（RPC）的方式得到 SensorManager。这样通过在类 SensorManager 中的方法，可以得到底层的各种传感器数据。上述方法的具体实现代码如下所示。

```
public int getSensors() {
    return getLegacySensorManager().getSensors();
}
public List<Sensor> getSensorList(int type) {
    List<Sensor> list;
    final List<Sensor> fullList = getFullSensorList();
    synchronized (mSensorListByType) {
        list = mSensorListByType.get(type);
        if (list == null) {
            if (type == Sensor.TYPE_ALL) {
                list = fullList;
            } else {
                list = new ArrayList<Sensor>();
                for (Sensor i : fullList) {
                    if (i.getType() == type)
                        list.add(i);
                }
            }
            list = Collections.unmodifiableList(list);
            mSensorListByType.append(type, list);
        }
    }
    return list;
}
public Sensor getDefaultSensor(int type) {
    List<Sensor> l = getSensorList(type);
    return l.isEmpty() ? null : l.get(0);
}
@Deprecated
public boolean registerListener(SensorListener listener, int sensors) {
    return registerListener(listener, sensors, SENSOR_DELAY_NORMAL);
}
@Deprecated
public boolean registerListener(SensorListener listener, int sensors, int rate) {
    return getLegacySensorManager().registerListener(listener, sensors, rate);
}
@Deprecated
public void unregisterListener(SensorListener listener) {
    unregisterListener(listener, SENSOR_ALL | SENSOR_ORIENTATION_RAW);
}
@Deprecated
public void unregisterListener(SensorListener listener, int sensors) {
    getLegacySensorManager().unregisterListener(listener, sensors);
}
public void unregisterListener(SensorEventListener listener, Sensor sensor) {
    if (listener == null || sensor == null) {
        return;
```

```
        }
        unregisterListenerImpl(listener, sensor);
    }
    public void unregisterListener(SensorEventListener listener) {
        if (listener == null) {
            return;
        }
        unregisterListenerImpl(listener, null);
    }
    protected abstract void unregisterListenerImpl(SensorEventListener listener, Sensor sensor);
    public boolean registerListener(SensorEventListener listener, Sensor sensor, int rate) {
        return registerListener(listener, sensor, rate, null);
    }
    public boolean registerListener(SensorEventListener listener, Sensor sensor, int rate,
            Handler handler) {
        if (listener == null || sensor == null) {
            return false;
        }

        int delay = -1;
        switch (rate) {
            case SENSOR_DELAY_FASTEST:
                delay = 0;
                break;
            case SENSOR_DELAY_GAME:
                delay = 20000;
                break;
            case SENSOR_DELAY_UI:
                delay = 66667;
                break;
            case SENSOR_DELAY_NORMAL:
                delay = 200000;
                break;
            default:
                delay = rate;
                break;
        }
        return registerListenerImpl(listener, sensor, delay, handler);
    }
    protected abstract boolean registerListenerImpl(SensorEventListener listener, Sensor sensor,
            int delay, Handler handler);

    public static boolean getRotationMatrix(float[] R, float[] I, float[] gravity, float[] geomagnetic) {
        float Ax = gravity[0];
        float Ay = gravity[1];
        float Az = gravity[2];
        final float Ex = geomagnetic[0];
        final float Ey = geomagnetic[1];
        final float Ez = geomagnetic[2];
        float Hx = Ey*Az - Ez*Ay;
        float Hy = Ez*Ax - Ex*Az;
```

```
        float Hz = Ex*Ay - Ey*Ax;
        final float normH = (float)Math.sqrt(Hx*Hx + Hy*Hy + Hz*Hz);
        if (normH < 0.1f) {
            return false;
        }
        final float invH = 1.0f / normH;
        Hx *= invH;
        Hy *= invH;
        Hz *= invH;
        final float invA = 1.0f / (float)Math.sqrt(Ax*Ax + Ay*Ay + Az*Az);
        Ax *= invA;
        Ay *= invA;
        Az *= invA;
        final float Mx = Ay*Hz - Az*Hy;
        final float My = Az*Hx - Ax*Hz;
        final float Mz = Ax*Hy - Ay*Hx;
        if (R != null) {
            if (R.length == 9) {
                R[0] = Hx;          R[1] = Hy;       R[2] = Hz;
                R[3] = Mx;          R[4] = My;       R[5] = Mz;
                R[6] = Ax;          R[7] = Ay;       R[8] = Az;
            } else if (R.length ==  16) {
                R[0]   = Hx;        R[1]   = Hy;     R[2]   = Hz;     R[3]   = 0;
                R[4]   = Mx;        R[5]   = My;     R[6]   = Mz;     R[7]   = 0;
                R[8]   = Ax;        R[9]   = Ay;     R[10] = Az;      R[11] = 0;
                R[12] = 0;          R[13] = 0;       R[14] = 0;       R[15] = 1;
            }
        }
        if (I != null) {
            final float invE = 1.0f / (float)Math.sqrt(Ex*Ex + Ey*Ey + Ez*Ez);
            final float c = (Ex*Mx + Ey*My + Ez*Mz) * invE;
            final float s = (Ex*Ax + Ey*Ay + Ez*Az) * invE;
            if (I.length == 9) {
                I[0] = 1;       I[1] = 0;       I[2] = 0;
                I[3] = 0;       I[4] = c;       I[5] = s;
                I[6] = 0;       I[7] =-s;       I[8] = c;
            } else if (I.length == 16) {
                I[0] = 1;       I[1] = 0;       I[2] = 0;
                I[4] = 0;       I[5] = c;       I[6] = s;
                I[8] = 0;       I[9] =-s;       I[10]= c;
                I[3] = I[7] = I[11] = I[12] = I[13] = I[14] = 0;
                I[15] = 1;
            }
        }
        return true;
    }
    public static float getInclination(float[] I) {
        if (I.length == 9) {
            return (float)Math.atan2(I[5], I[4]);
        } else {
            return (float)Math.atan2(I[6], I[5]);
```

```
        }
    }

    public static boolean remapCoordinateSystem(float[] inR, int X, int Y, float[] outR)
    {
        if (inR == outR) {
            final float[] temp = mTempMatrix;
            synchronized(temp) {
                if (remapCoordinateSystemImpl(inR, X, Y, temp)) {
                    final int size = outR.length;
                    for (int i=0 ; i<size ; i++)
                        outR[i] = temp[i];
                    return true;
                }
            }
        }
        return remapCoordinateSystemImpl(inR, X, Y, outR);
    }

    private static boolean remapCoordinateSystemImpl(float[] inR, int X, int Y, float[] outR)
    {

        final int length = outR.length;
        if (inR.length != length)
            return false;     // invalid parameter
        if ((X & 0x7C)!=0 || (Y & 0x7C)!=0)
            return false;     // invalid parameter
        if (((X & 0x3)==0) || ((Y & 0x3)==0))
            return false;     // no axis specified
        if ((X & 0x3) == (Y & 0x3))
            return false;     // same axis specified
        int Z = X ^ Y;

        final int x = (X & 0x3)-1;
        final int y = (Y & 0x3)-1;
        final int z = (Z & 0x3)-1;

        final int axis_y = (z+1)%3;
        final int axis_z = (z+2)%3;
        if (((x^axis_y)|(y^axis_z)) != 0)
            Z ^= 0x80;

        final boolean sx = (X>=0x80);
        final boolean sy = (Y>=0x80);
        final boolean sz = (Z>=0x80);

        final int rowLength = ((length==16)?4:3);
        for (int j=0 ; j<3 ; j++) {
            final int offset = j*rowLength;
            for (int i=0 ; i<3 ; i++) {
                if (x==i)     outR[offset+i] = sx ? -inR[offset+0] : inR[offset+0];
```

```
                if (y==i)     outR[offset+i] = sy ? -inR[offset+1] : inR[offset+1];
                if (z==i)     outR[offset+i] = sz ? -inR[offset+2] : inR[offset+2];
            }
        }
        if (length == 16) {
            outR[3] = outR[7] = outR[11] = outR[12] = outR[13] = outR[14] = 0;
            outR[15] = 1;
        }
        return true;
    }
    public static float[] getOrientation(float[] R, float values[]) {

        if (R.length == 9) {
            values[0] = (float)Math.atan2(R[1], R[4]);
            values[1] = (float)Math.asin(-R[7]);
            values[2] = (float)Math.atan2(-R[6], R[8]);
        } else {
            values[0] = (float)Math.atan2(R[1], R[5]);
            values[1] = (float)Math.asin(-R[9]);
            values[2] = (float)Math.atan2(-R[8], R[10]);
        }
        return values;
    }
    public static float getAltitude(float p0, float p) {
        final float coef = 1.0f / 12.255f;
        return 44330.0f * (1.0f - (float)Math.pow(p/p0, coef));
    }

    public static void getAngleChange( float[] angleChange, float[] R, float[] prevR) {
        float rd1=0,rd4=0, rd6=0,rd7=0, rd8=0;
        float ri0=0,ri1=0,ri2=0,ri3=0,ri4=0,ri5=0,ri6=0,ri7=0,ri8=0;
        float pri0=0, pri1=0, pri2=0, pri3=0, pri4=0, pri5=0, pri6=0, pri7=0, pri8=0;

        if(R.length == 9) {
            ri0 = R[0];
            ri1 = R[1];
            ri2 = R[2];
            ri3 = R[3];
            ri4 = R[4];
            ri5 = R[5];
            ri6 = R[6];
            ri7 = R[7];
            ri8 = R[8];
        } else if(R.length == 16) {
            ri0 = R[0];
            ri1 = R[1];
            ri2 = R[2];
            ri3 = R[4];
            ri4 = R[5];
            ri5 = R[6];
            ri6 = R[8];
```

```
            ri7 = R[9];
            ri8 = R[10];
        }

        if(prevR.length == 9) {
            pri0 = prevR[0];
            pri1 = prevR[1];
            pri2 = prevR[2];
            pri3 = prevR[3];
            pri4 = prevR[4];
            pri5 = prevR[5];
            pri6 = prevR[6];
            pri7 = prevR[7];
            pri8 = prevR[8];
        } else if(prevR.length == 16) {
            pri0 = prevR[0];
            pri1 = prevR[1];
            pri2 = prevR[2];
            pri3 = prevR[4];
            pri4 = prevR[5];
            pri5 = prevR[6];
            pri6 = prevR[8];
            pri7 = prevR[9];
            pri8 = prevR[10];
        }
        rd1 = pri0 * ri1 + pri3 * ri4 + pri6 * ri7; //rd[0][1]
        rd4 = pri1 * ri1 + pri4 * ri4 + pri7 * ri7; //rd[1][1]
        rd6 = pri2 * ri0 + pri5 * ri3 + pri8 * ri6; //rd[2][0]
        rd7 = pri2 * ri1 + pri5 * ri4 + pri8 * ri7; //rd[2][1]
        rd8 = pri2 * ri2 + pri5 * ri5 + pri8 * ri8; //rd[2][2]

        angleChange[0] = (float)Math.atan2(rd1, rd4);
        angleChange[1] = (float)Math.asin(-rd7);
        angleChange[2] = (float)Math.atan2(-rd6, rd8);
    }
    public static void getRotationMatrixFromVector(float[] R, float[] rotationVector) {

        float q0;
        float q1 = rotationVector[0];
        float q2 = rotationVector[1];
        float q3 = rotationVector[2];

        if (rotationVector.length == 4) {
            q0 = rotationVector[3];
        } else {
            q0 = 1 - q1*q1 - q2*q2 - q3*q3;
            q0 = (q0 > 0) ? (float)Math.sqrt(q0) : 0;
        }
```

```
        float sq_q1 = 2 * q1 * q1;
        float sq_q2 = 2 * q2 * q2;
        float sq_q3 = 2 * q3 * q3;
        float q1_q2 = 2 * q1 * q2;
        float q3_q0 = 2 * q3 * q0;
        float q1_q3 = 2 * q1 * q3;
        float q2_q0 = 2 * q2 * q0;
        float q2_q3 = 2 * q2 * q3;
        float q1_q0 = 2 * q1 * q0;

        if(R.length == 9) {
            R[0] = 1 - sq_q2 - sq_q3;
            R[1] = q1_q2 - q3_q0;
            R[2] = q1_q3 + q2_q0;

            R[3] = q1_q2 + q3_q0;
            R[4] = 1 - sq_q1 - sq_q3;
            R[5] = q2_q3 - q1_q0;

            R[6] = q1_q3 - q2_q0;
            R[7] = q2_q3 + q1_q0;
            R[8] = 1 - sq_q1 - sq_q2;
        } else if (R.length == 16) {
            R[0] = 1 - sq_q2 - sq_q3;
            R[1] = q1_q2 - q3_q0;
            R[2] = q1_q3 + q2_q0;
            R[3] = 0.0f;

            R[4] = q1_q2 + q3_q0;
            R[5] = 1 - sq_q1 - sq_q3;
            R[6] = q2_q3 - q1_q0;
            R[7] = 0.0f;

            R[8] = q1_q3 - q2_q0;
            R[9] = q2_q3 + q1_q0;
            R[10] = 1 - sq_q1 - sq_q2;
            R[11] = 0.0f;

            R[12] = R[13] = R[14] = 0.0f;
            R[15] = 1.0f;
        }
    }
    public static void getQuaternionFromVector(float[] Q, float[] rv) {
        if (rv.length == 4) {
            Q[0] = rv[3];
        } else {
            Q[0] = 1 - rv[0]*rv[0] - rv[1]*rv[1] - rv[2]*rv[2];
            Q[0] = (Q[0] > 0) ? (float)Math.sqrt(Q[0]) : 0;
```

```
            }
            Q[1] = rv[0];
            Q[2] = rv[1];
            Q[3] = rv[2];
        }
        public boolean requestTriggerSensor(TriggerEventListener listener, Sensor sensor) {
            return requestTriggerSensorImpl(listener, sensor);
        }
        protected abstract boolean requestTriggerSensorImpl(TriggerEventListener listener, Sensor sensor);
        public boolean cancelTriggerSensor(TriggerEventListener listener, Sensor sensor) {
            return cancelTriggerSensorImpl(listener, sensor, true);
        protected abstract boolean cancelTriggerSensorImpl(TriggerEventListener listener,
                Sensor sensor, boolean disable);
        private LegacySensorManager getLegacySensorManager() {
            synchronized (mSensorListByType) {
                if (mLegacySensorManager == null) {
                    Log.i(TAG, "This application is using deprecated SensorManager API which will "
                            + "be removed someday.   Please consider switching to the new API.");
                    mLegacySensorManager = new LegacySensorManager(this);
                }
                return mLegacySensorManager;
            }
        }
}
```

上述代码的方法其实就是在开发传感器应用程序时用到的 API 接口。有关上述方法的详细介绍，读者可以查阅官网 SDK API 中关于类 android.hardware.SensorManager 的具体说明，如图 11-3 所示。

- Introduction
- App Components
- App Resources
- App Manifest
- User Interface
- Animation and Graphics
- Computation
- Media and Camera
- Location and Sensors
- Connectivity
- Text and Input
- Data Storage
- Administration

Sensor	Type	Description	Common Uses
TYPE_ACCELEROMETER	Hardware	Measures the acceleration force in m/s^2 that is applied to a device on all three physical axes (x, y, and z), including the force of gravity.	Motion detection (shake, tilt, etc.).
TYPE_AMBIENT_TEMPERATURE	Hardware	Measures the ambient room temperature in degrees Celsius (°C). See note below.	Monitoring air temperatures.
TYPE_GRAVITY	Software or Hardware	Measures the force of gravity in m/s^2 that is applied to a device on all three physical axes (x, y, z).	Motion detection (shake, tilt, etc.).
TYPE_GYROSCOPE	Hardware	Measures a device's rate of rotation in rad/s around each of the three physical axes (x, y, and z).	Rotation detection (spin, turn, etc.).
TYPE_LIGHT	Hardware	Measures the ambient light level (illumination) in lx.	Controlling screen brightness.
TYPE_LINEAR_ACCELERATION	Software or Hardware	Measures the acceleration force in m/s^2 that is applied to a device on all three physical axes (x, y, and z), excluding the force of	Monitoring acceleration along a single

图 11-3　SDK API 中对于类 android.hardware.SensorManager 的具体说明

11.4　JNI 层详解

在 Android 系统中，传感器系统的 JNI 部分的代码路径是 frameworks/base/core/jni/android_hardware_SensorManager.cpp。

在此文件中提供了对类 android.hardware.Sensor.Manager 的本地支持。上层和 JNI 层的调用关系如图 11-4 所示。

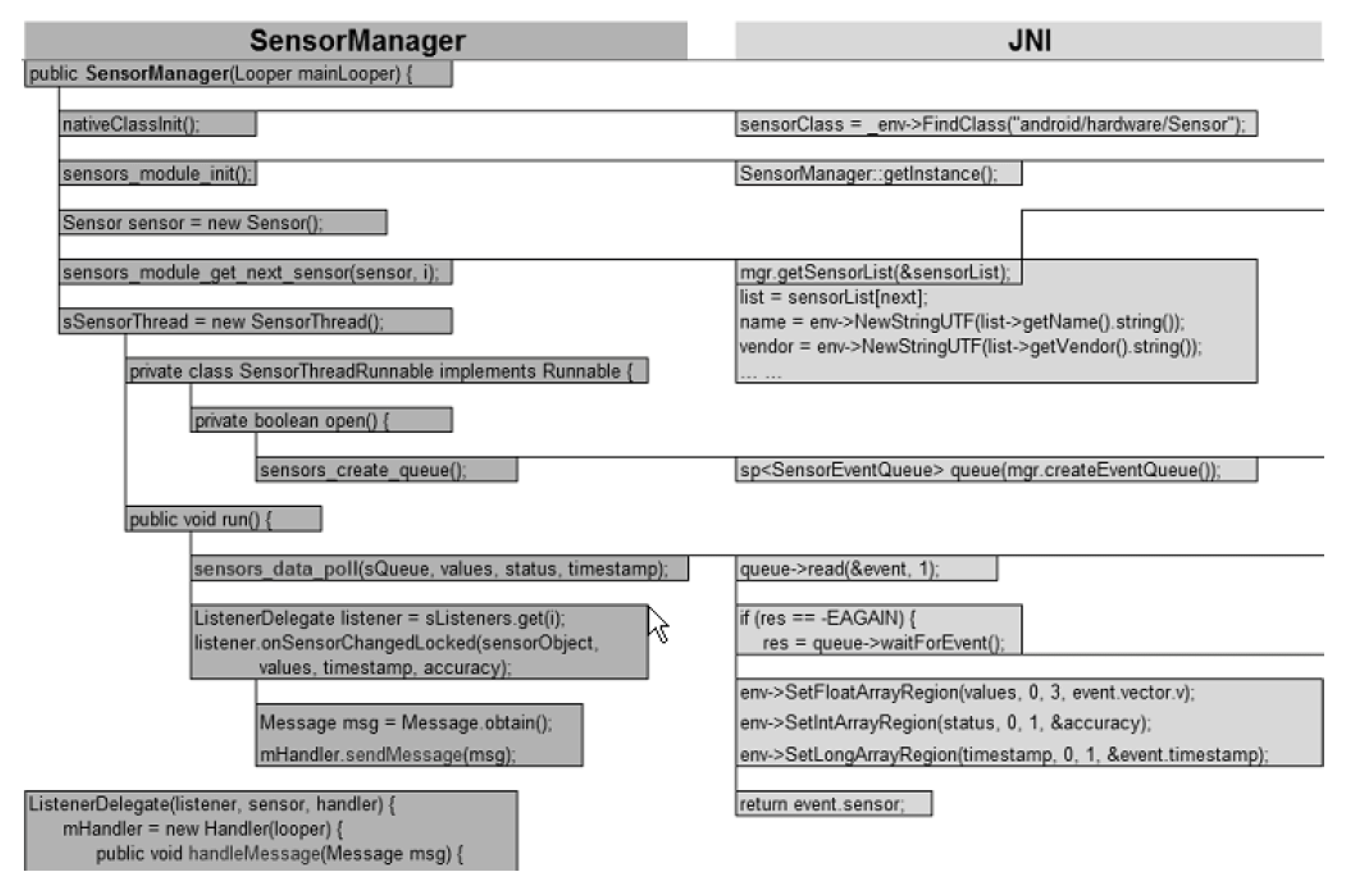


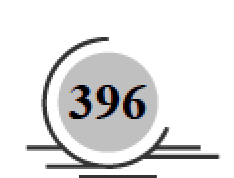
图 11-4　上层和 JNI 层的调用关系

在如图 11-4 所示的调用关系中涉及了如下 API 接口方法。

☑ nativeClassInit()：在 JNI 层得到 android.hardware.Sensor 的 JNI 环境指针。
☑ sensors_module_init()：通过 JNI 调用本地框架，得到 SensorService，SensorService 初始化控制流各功能。
☑ new Sensor()：建立一个 Sensor 对象，具体可查阅官网 API 中的 android.hardware.Sensor 部分。
☑ sensors_module_get_next_sensor()：上层得到设备支持的所有 Sensor，并放入 SensorList 链表。
☑ new SensorThread()：创建 Sensor 线程，当应用程序 registerListener()注册监听器时开启线程 run()，注意当没有数据变化时线程会阻塞。

11.4.1　实现 Native（本地）函数

文件 android_hardware_SensorManager.cpp 的功能是实现文件 SensorManager.java 中的 Native（本



地）函数，主要是通过调用文件 SenrsorManager.cpp 和 SensorEventQueue.cpp 中的相关类来完成相关工作的。文件 android_hardware_SensorManager.cpp 的具体实现代码如下所示。

```
static struct {
    jclass clazz;
    jmethodID dispatchSensorEvent;
} gBaseEventQueueClassInfo;

namespace android {

struct SensorOffsets
{
    jfieldID name;
    jfieldID vendor;
    jfieldID version;
    jfieldID handle;
    jfieldID type;
    jfieldID range;
    jfieldID resolution;
    jfieldID power;
    jfieldID minDelay;
} gSensorOffsets;


/*
 * The method below are not thread-safe and not intended to be
 */

static void
nativeClassInit (JNIEnv *_env, jclass _this)
{
    jclass sensorClass = _env->FindClass("android/hardware/Sensor");
    SensorOffsets& sensorOffsets = gSensorOffsets;
    sensorOffsets.name = _env->GetFieldID(sensorClass, "mName", "Ljava/lang/String;");
    sensorOffsets.vendor = _env->GetFieldID(sensorClass, "mVendor", "Ljava/lang/String;");
    sensorOffsets.version = _env->GetFieldID(sensorClass, "mVersion", "I");
    sensorOffsets.handle = _env->GetFieldID(sensorClass, "mHandle", "I");
    sensorOffsets.type = _env->GetFieldID(sensorClass, "mType", "I");
    sensorOffsets.range = _env->GetFieldID(sensorClass, "mMaxRange", "F");
    sensorOffsets.resolution = _env->GetFieldID(sensorClass, "mResolution","F");
    sensorOffsets.power = _env->GetFieldID(sensorClass, "mPower", "F");
    sensorOffsets.minDelay = _env->GetFieldID(sensorClass, "mMinDelay", "I");
}

static jint
nativeGetNextSensor(JNIEnv *env, jclass clazz, jobject sensor, jint next)
{
    SensorManager& mgr(SensorManager::getInstance());

    Sensor const* const* sensorList;
```

```
    size_t count = mgr.getSensorList(&sensorList);
    if (size_t(next) >= count)
        return -1;

    Sensor const* const list = sensorList[next];
    const SensorOffsets& sensorOffsets(gSensorOffsets);
    jstring name = env->NewStringUTF(list->getName().string());
    jstring vendor = env->NewStringUTF(list->getVendor().string());
    env->SetObjectField(sensor, sensorOffsets.name, name);
    env->SetObjectField(sensor, sensorOffsets.vendor, vendor);
    env->SetIntField(sensor, sensorOffsets.version, list->getVersion());
    env->SetIntField(sensor, sensorOffsets.handle, list->getHandle());
    env->SetIntField(sensor, sensorOffsets.type, list->getType());
    env->SetFloatField(sensor, sensorOffsets.range, list->getMaxValue());
    env->SetFloatField(sensor, sensorOffsets.resolution, list->getResolution());
    env->SetFloatField(sensor, sensorOffsets.power, list->getPowerUsage());
    env->SetIntField(sensor, sensorOffsets.minDelay, list->getMinDelay());

    next++;
    return size_t(next) < count ? next : 0;
}

//----------------------------------------------------------------------------

class Receiver : public LooperCallback {
    sp<SensorEventQueue> mSensorQueue;
    sp<MessageQueue> mMessageQueue;
    jobject mReceiverObject;
    jfloatArray mScratch;
public:
    Receiver(const sp<SensorEventQueue>& sensorQueue,
            const sp<MessageQueue>& messageQueue,
            jobject receiverObject, jfloatArray scratch) {
        JNIEnv* env = AndroidRuntime::getJNIEnv();
        mSensorQueue = sensorQueue;
        mMessageQueue = messageQueue;
        mReceiverObject = env->NewGlobalRef(receiverObject);
        mScratch = (jfloatArray)env->NewGlobalRef(scratch);
    }
    ~Receiver() {
        JNIEnv* env = AndroidRuntime::getJNIEnv();
        env->DeleteGlobalRef(mReceiverObject);
        env->DeleteGlobalRef(mScratch);
    }
    sp<SensorEventQueue> getSensorEventQueue() const {
        return mSensorQueue;
    }

    void destroy() {
        mMessageQueue->getLooper()->removeFd( mSensorQueue->getFd() );
    }
```

```
private:
    virtual void onFirstRef() {
        LooperCallback::onFirstRef();
        mMessageQueue->getLooper()->addFd(mSensorQueue->getFd(), 0,
                ALOOPER_EVENT_INPUT, this, mSensorQueue.get());
    }

    virtual int handleEvent(int fd, int events, void* data) {
        JNIEnv* env = AndroidRuntime::getJNIEnv();
        sp<SensorEventQueue> q = reinterpret_cast<SensorEventQueue *>(data);
        ssize_t n;
        ASensorEvent buffer[16];
        while ((n = q->read(buffer, 16)) > 0) {
            for (int i=0 ; i<n ; i++) {

                env->SetFloatArrayRegion(mScratch, 0, 16, buffer[i].data);

                env->CallVoidMethod(mReceiverObject,
                        gBaseEventQueueClassInfo.dispatchSensorEvent,
                        buffer[i].sensor,
                        mScratch,
                        buffer[i].vector.status,
                        buffer[i].timestamp);

                if (env->ExceptionCheck()) {
                    ALOGE("Exception dispatching input event.");
                    return 1;
                }
            }
        }
        if (n<0 && n != -EAGAIN) {
        }

        return 1;
    }
};

static jint nativeInitSensorEventQueue(JNIEnv *env, jclass clazz, jobject eventQ, jobject msgQ, jfloatArray
scratch) {
    SensorManager& mgr(SensorManager::getInstance());
    sp<SensorEventQueue> queue(mgr.createEventQueue());

    sp<MessageQueue> messageQueue = android_os_MessageQueue_getMessageQueue(env, msgQ);
    if (messageQueue == NULL) {
        jniThrowRuntimeException(env, "MessageQueue is not initialized.");
        return 0;
    }

    sp<Receiver> receiver = new Receiver(queue, messageQueue, eventQ, scratch);
    receiver->incStrong((void*)nativeInitSensorEventQueue);
```

```
    return jint(receiver.get());
}

static jint nativeEnableSensor(JNIEnv *env, jclass clazz, jint eventQ, jint handle, jint us) {
    sp<Receiver> receiver(reinterpret_cast<Receiver *>(eventQ));
    return receiver->getSensorEventQueue()->enableSensor(handle, us);
}

static jint nativeDisableSensor(JNIEnv *env, jclass clazz, jint eventQ, jint handle) {
    sp<Receiver> receiver(reinterpret_cast<Receiver *>(eventQ));
    return receiver->getSensorEventQueue()->disableSensor(handle);
}

static void nativeDestroySensorEventQueue(JNIEnv *env, jclass clazz, jint eventQ, jint handle) {
    sp<Receiver> receiver(reinterpret_cast<Receiver *>(eventQ));
    receiver->destroy();
    receiver->decStrong((void*)nativeInitSensorEventQueue);
}


//----------------------------------------------------------------------------

static JNINativeMethod gSystemSensorManagerMethods[] = {
    {"nativeClassInit",
            "()V",
            (void*)nativeClassInit },

    {"nativeGetNextSensor",
            "(Landroid/hardware/Sensor;I)I",
            (void*)nativeGetNextSensor },
};

static JNINativeMethod gBaseEventQueueMethods[] = {
    {"nativeInitBaseEventQueue",

"(Landroid/hardware/SystemSensorManager$BaseEventQueue;Landroid/os/MessageQueue;[F)I",
            (void*)nativeInitSensorEventQueue },

    {"nativeEnableSensor",
            "(III)I",
            (void*)nativeEnableSensor },

    {"nativeDisableSensor",
            "(II)I",
            (void*)nativeDisableSensor },

    {"nativeDestroySensorEventQueue",
            "(I)V",
            (void*)nativeDestroySensorEventQueue },
};
```

```
};

using namespace android;

#define FIND_CLASS(var, className) \
        var = env->FindClass(className); \
        LOG_FATAL_IF(! var, "Unable to find class " className); \
        var = jclass(env->NewGlobalRef(var));

#define GET_METHOD_ID(var, clazz, methodName, methodDescriptor) \
        var = env->GetMethodID(clazz, methodName, methodDescriptor); \
        LOG_FATAL_IF(! var, "Unable to find method " methodName);

int register_android_hardware_SensorManager(JNIEnv *env)
{
    jniRegisterNativeMethods(env, "android/hardware/SystemSensorManager",
            gSystemSensorManagerMethods, NELEM(gSystemSensorManagerMethods));

    jniRegisterNativeMethods(env, "android/hardware/SystemSensorManager$BaseEventQueue",
            gBaseEventQueueMethods, NELEM(gBaseEventQueueMethods));

    FIND_CLASS(gBaseEventQueueClassInfo.clazz,
"android/hardware/SystemSensorManager$BaseEventQueue");

    GET_METHOD_ID(gBaseEventQueueClassInfo.dispatchSensorEvent,
            gBaseEventQueueClassInfo.clazz,
            "dispatchSensorEvent", "(I[FIJ)V");

    return 0;
}
```

11.4.2　处理客户端数据

文件 frameworks/native/libs/gui/SensorManager.cpp 的功能是提供了对传感器数据部分的操作，实现了 sensor_data_XXX()格式的函数。另外在 Native 层的客户端，文件 SensorManager.cpp 还负责与服务端 SensorService.cpp 之间的通信工作。文件 SensorManager.cpp 的具体实现代码如下所示。

```
// ----------------------------------------------------------------------------
namespace android {
// ----------------------------------------------------------------------------

ANDROID_SINGLETON_STATIC_INSTANCE(SensorManager)

SensorManager::SensorManager()
    : mSensorList(0)
{
    // okay we're not locked here, but it's not needed during construction
    assertStateLocked();
}
```

```
SensorManager::~SensorManager()
{
    free(mSensorList);
}

void SensorManager::sensorManagerDied()
{
    Mutex::Autolock _l(mLock);
    mSensorServer.clear();
    free(mSensorList);
    mSensorList = NULL;
    mSensors.clear();
}

status_t SensorManager::assertStateLocked() const {
    if (mSensorServer == NULL) {
        const String16 name("sensorservice");
        for (int i=0 ; i<4 ; i++) {
            status_t err = getService(name, &mSensorServer);
            if (err == NAME_NOT_FOUND) {
                usleep(250000);
                continue;
            }
            if (err != NO_ERROR) {
                return err;
            }
            break;
        }

        class DeathObserver : public IBinder::DeathRecipient {
            SensorManager& mSensorManger;
            virtual void binderDied(const wp<IBinder>& who) {
                ALOGW("sensorservice died [%p]", who.unsafe_get());
                mSensorManger.sensorManagerDied();
            }
        public:
            DeathObserver(SensorManager& mgr) : mSensorManger(mgr) { }
        };

        mDeathObserver = new DeathObserver(*const_cast<SensorManager *>(this));
        mSensorServer->asBinder()->linkToDeath(mDeathObserver);

        mSensors = mSensorServer->getSensorList();
        size_t count = mSensors.size();
        mSensorList = (Sensor const**)malloc(count * sizeof(Sensor*));
        for (size_t i=0 ; i<count ; i++) {
            mSensorList[i] = mSensors.array() + i;
        }
    }

    return NO_ERROR;
```

```
}


ssize_t SensorManager::getSensorList(Sensor const* const** list) const
{
    Mutex::Autolock _l(mLock);
    status_t err = assertStateLocked();
    if (err < 0) {
        return ssize_t(err);
    }
    *list = mSensorList;
    return mSensors.size();
}

Sensor const* SensorManager::getDefaultSensor(int type)
{
    Mutex::Autolock _l(mLock);
    if (assertStateLocked() == NO_ERROR) {
        for (size_t i=0 ; i<mSensors.size() ; i++) {
            if (mSensorList[i]->getType() == type)
                return mSensorList[i];
        }
    }
    return NULL;
}

sp<SensorEventQueue> SensorManager::createEventQueue()
{
    sp<SensorEventQueue> queue;

    Mutex::Autolock _l(mLock);
    while (assertStateLocked() == NO_ERROR) {
        sp<ISensorEventConnection> connection =
                mSensorServer->createSensorEventConnection();
        if (connection == NULL) {
            ALOGE("createEventQueue: connection is NULL. SensorService died.");
            continue;
        }
        queue = new SensorEventQueue(connection);
        break;
    }
    return queue;
}

// ----------------------------------------------------------------------------
};
```

11.4.3　处理服务端数据

文件 frameworks/native/services/sensorservice/SensorService.cpp 的功能是实现了 Sensor 真正的后台

服务，是服务端的数据处理中心。在 Android 的传感器系统中，SensorService 作为一个轻量级的 System Service 运行于 SystemServer 内，即在 system_init<system_init.cpp>中调用了 SensorService::instantiate()。SensorService 的主要功能如下所示。

（1）通过 SensorService::instantiate 创建实例对象，并增加到 ServiceManager 中，然后创建并启动线程，执行 threadLoop。

（2）threadLoop 从 Sensor 驱动获取原始数据，然后通过 SensorEventConnection 把事件发送给客户端。

（3）BnSensorServer 的成员函数负责让客户端获取 Sensor 列表和创建 SensorEventConnection。

文件 SensorService.cpp 的具体实现代码如下所示。

```
namespace android {

const char* SensorService::WAKE_LOCK_NAME = "SensorService";

SensorService::SensorService()
    : mInitCheck(NO_INIT)
{
}

void SensorService::onFirstRef()
{
    ALOGD("nuSensorService starting...");

    SensorDevice& dev(SensorDevice::getInstance());

    if (dev.initCheck() == NO_ERROR) {
        sensor_t const* list;
        ssize_t count = dev.getSensorList(&list);
        if (count > 0) {
            ssize_t orientationIndex = -1;
            bool hasGyro = false;
            uint32_t virtualSensorsNeeds =
                    (1<<SENSOR_TYPE_GRAVITY) |
                    (1<<SENSOR_TYPE_LINEAR_ACCELERATION) |
                    (1<<SENSOR_TYPE_ROTATION_VECTOR);

            mLastEventSeen.setCapacity(count);
            for (ssize_t i=0 ; i<count ; i++) {
                registerSensor( new HardwareSensor(list[i]) );
                switch (list[i].type) {
                    case SENSOR_TYPE_ORIENTATION:
                        orientationIndex = i;
                        break;
                    case SENSOR_TYPE_GYROSCOPE:
                        hasGyro = true;
                        break;
                    case SENSOR_TYPE_GRAVITY:
```

```
                    case SENSOR_TYPE_LINEAR_ACCELERATION:
                    case SENSOR_TYPE_ROTATION_VECTOR:
                        virtualSensorsNeeds &= ~(1<<list[i].type);
                        break;
                }
            }
            const SensorFusion& fusion(SensorFusion::getInstance());

            if (hasGyro) {
                registerVirtualSensor( new RotationVectorSensor() );
                registerVirtualSensor( new GravitySensor(list, count) );
                registerVirtualSensor( new LinearAccelerationSensor(list, count) );

                registerVirtualSensor( new OrientationSensor() );
                registerVirtualSensor( new CorrectedGyroSensor(list, count) );
            }
            mUserSensorList = mSensorList;

            if (hasGyro) {
                registerVirtualSensor( new GyroDriftSensor() );
            }

            if (hasGyro &&
                    (virtualSensorsNeeds & (1<<SENSOR_TYPE_ROTATION_VECTOR))) {
                if (orientationIndex >= 0) {
                    mUserSensorList.removeItemsAt(orientationIndex);
                }
            }
            for (size_t i=0 ; i<mSensorList.size() ; i++) {
                switch (mSensorList[i].getType()) {
                    case SENSOR_TYPE_GRAVITY:
                    case SENSOR_TYPE_LINEAR_ACCELERATION:
                    case SENSOR_TYPE_ROTATION_VECTOR:
                        if (strstr(mSensorList[i].getVendor().string(), "Google")) {
                            mUserSensorListDebug.add(mSensorList[i]);
                        }
                        break;
                    default:
                        mUserSensorListDebug.add(mSensorList[i]);
                        break;
                }
            }

            run("SensorService", PRIORITY_URGENT_DISPLAY);
            mInitCheck = NO_ERROR;
        }
    }
}
```

```
void SensorService::registerSensor(SensorInterface* s)
{
    sensors_event_t event;
    memset(&event, 0, sizeof(event));

    const Sensor sensor(s->getSensor());
    mSensorList.add(sensor);
    mSensorMap.add(sensor.getHandle(), s);
    mLastEventSeen.add(sensor.getHandle(), event);
}

void SensorService::registerVirtualSensor(SensorInterface* s)
{
    registerSensor(s);
    mVirtualSensorList.add( s );
}

SensorService::~SensorService()
{
    for (size_t i=0 ; i<mSensorMap.size() ; i++)
        delete mSensorMap.valueAt(i);
}

static const String16 sDump("android.permission.DUMP");

status_t SensorService::dump(int fd, const Vector<String16>& args)
{
    const size_t SIZE = 1024;
    char buffer[SIZE];
    String8 result;
    if (!PermissionCache::checkCallingPermission(sDump)) {
        snprintf(buffer, SIZE, "Permission Denial: "
                "can't dump SurfaceFlinger from pid=%d, uid=%d\n",
                IPCThreadState::self()->getCallingPid(),
                IPCThreadState::self()->getCallingUid());
        result.append(buffer);
    } else {
        Mutex::Autolock _l(mLock);
        snprintf(buffer, SIZE, "Sensor List:\n");
        result.append(buffer);
        for (size_t i=0 ; i<mSensorList.size() ; i++) {
            const Sensor& s(mSensorList[i]);
            const sensors_event_t& e(mLastEventSeen.valueFor(s.getHandle()));
            snprintf(buffer, SIZE,
                    "%-48s| %-32s | 0x%08x | maxRate=%7.2fHz | "
                    "last=<%12.1f,%12.1f,%12.1f>\n",
                    s.getName().string(),
                    s.getVendor().string(),
                    s.getHandle(),
```

```
                        s.getMinDelay() ? (1000000.0f / s.getMinDelay()) : 0.0f,
                        e.data[0], e.data[1], e.data[2]);
                result.append(buffer);
            }
            SensorFusion::getInstance().dump(result, buffer, SIZE);
            SensorDevice::getInstance().dump(result, buffer, SIZE);

            snprintf(buffer, SIZE, "%d active connections\n",
                    mActiveConnections.size());
            result.append(buffer);
            snprintf(buffer, SIZE, "Active sensors:\n");
            result.append(buffer);
            for (size_t i=0 ; i<mActiveSensors.size() ; i++) {
                int handle = mActiveSensors.keyAt(i);
                snprintf(buffer, SIZE, "%s (handle=0x%08x, connections=%d)\n",
                        getSensorName(handle).string(),
                        handle,
                        mActiveSensors.valueAt(i)->getNumConnections());
                result.append(buffer);
            }
        }
        write(fd, result.string(), result.size());
        return NO_ERROR;
}

void SensorService::cleanupAutoDisabledSensor(const sp<SensorEventConnection>& connection,
            sensors_event_t const* buffer, const int count) {
        SensorInterface* sensor;
        status_t err = NO_ERROR;
        for (int i=0 ; i<count ; i++) {
            int handle = buffer[i].sensor;
            if (getSensorType(handle) == SENSOR_TYPE_SIGNIFICANT_MOTION) {
                if (connection->hasSensor(handle)) {
                    sensor = mSensorMap.valueFor(handle);
                    err = sensor ?sensor->resetStateWithoutActuatingHardware(connection.get(), handle)
                            : status_t(BAD_VALUE);
                    if (err != NO_ERROR) {
                        ALOGE("Sensor Inteface: Resetting state failed with err: %d", err);
                    }
                    cleanupWithoutDisable(connection, handle);
                }
            }
        }
}

bool SensorService::threadLoop()
{
        ALOGD("nuSensorService thread starting...");
```

```
const size_t numEventMax = 16;
const size_t minBufferSize = numEventMax + numEventMax * mVirtualSensorList.size();
sensors_event_t buffer[minBufferSize];
sensors_event_t scratch[minBufferSize];
SensorDevice& device(SensorDevice::getInstance());
const size_t vcount = mVirtualSensorList.size();

ssize_t count;
bool wakeLockAcquired = false;
const int halVersion = device.getHalDeviceVersion();
do {
    count = device.poll(buffer, numEventMax);
    if (count<0) {
        ALOGE("sensor poll failed (%s)", strerror(-count));
        break;
    }

    for (int i = 0; i < count; i++) {
        if (getSensorType(buffer[i].sensor) == SENSOR_TYPE_SIGNIFICANT_MOTION) {
             acquire_wake_lock(PARTIAL_WAKE_LOCK, WAKE_LOCK_NAME);
             wakeLockAcquired = true;
             break;
        }
    }

    recordLastValue(buffer, count);

    if (count && vcount) {
        sensors_event_t const * const event = buffer;
        const DefaultKeyedVector<int, SensorInterface*> virtualSensors(
                getActiveVirtualSensors());
        const size_t activeVirtualSensorCount = virtualSensors.size();
        if (activeVirtualSensorCount) {
            size_t k = 0;
            SensorFusion& fusion(SensorFusion::getInstance());
            if (fusion.isEnabled()) {
                for (size_t i=0 ; i<size_t(count) ; i++) {
                    fusion.process(event[i]);
                }
            }
            for (size_t i=0 ; i<size_t(count) && k<minBufferSize ; i++) {
                for (size_t j=0 ; j<activeVirtualSensorCount ; j++) {
                    if (count + k >= minBufferSize) {
                        ALOGE("buffer too small to hold all events: "
                                "count=%u, k=%u, size=%u",
                                count, k, minBufferSize);
                        break;
                    }
                    sensors_event_t out;
```

```
                        SensorInterface* si = virtualSensors.valueAt(j);
                        if (si->process(&out, event[i])) {
                            buffer[count + k] = out;
                            k++;
                        }
                    }
                }
                if (k) {
                    recordLastValue(&buffer[count], k);
                    count += k;
                    // sort the buffer by time-stamps
                    sortEventBuffer(buffer, count);
                }
            }
        }

        if (halVersion < SENSORS_DEVICE_API_VERSION_1_0) {
            for (int i = 0; i < count; i++) {
                if (getSensorType(buffer[i].sensor) == SENSOR_TYPE_ROTATION_VECTOR) {
                    // All the 4 components of the quaternion should be available
                    // No heading accuracy. Set it to -1
                    buffer[i].data[4] = -1;
                }
            }
        }

        const SortedVector< wp<SensorEventConnection> > activeConnections(
                getActiveConnections());
        size_t numConnections = activeConnections.size();
        for (size_t i=0 ; i<numConnections ; i++) {
            sp<SensorEventConnection> connection(
                    activeConnections[i].promote());
            if (connection != 0) {
                connection->sendEvents(buffer, count, scratch);
                cleanupAutoDisabledSensor(connection, buffer, count);
            }
        }

        if (wakeLockAcquired) release_wake_lock(WAKE_LOCK_NAME);

    } while (count >= 0 || Thread::exitPending());

    ALOGW("Exiting SensorService::threadLoop => aborting...");
    abort();
    return false;
}

void SensorService::recordLastValue(
        sensors_event_t const * buffer, size_t count)
```

```
{
    Mutex::Autolock _l(mLock);

    int32_t prev = buffer[0].sensor;
    for (size_t i=1 ; i<count ; i++) {
        int32_t curr = buffer[i].sensor;
        if (curr != prev) {
            mLastEventSeen.editValueFor(prev) = buffer[i-1];
            prev = curr;
        }
    }
    mLastEventSeen.editValueFor(prev) = buffer[count-1];
}

void SensorService::sortEventBuffer(sensors_event_t* buffer, size_t count)
{
    struct compar {
        static int cmp(void const* lhs, void const* rhs) {
            sensors_event_t const* l = static_cast<sensors_event_t const*>(lhs);
            sensors_event_t const* r = static_cast<sensors_event_t const*>(rhs);
            return l->timestamp - r->timestamp;
        }
    };
    qsort(buffer, count, sizeof(sensors_event_t), compar::cmp);
}

SortedVector< wp<SensorService::SensorEventConnection> >
SensorService::getActiveConnections() const
{
    Mutex::Autolock _l(mLock);
    return mActiveConnections;
}

DefaultKeyedVector<int, SensorInterface*>
SensorService::getActiveVirtualSensors() const
{
    Mutex::Autolock _l(mLock);
    return mActiveVirtualSensors;
}

String SensorService::getSensorName(int handle) const {
    size_t count = mUserSensorList.size();
    for (size_t i=0 ; i<count ; i++) {
        const Sensor& sensor(mUserSensorList[i]);
        if (sensor.getHandle() == handle) {
            return sensor.getName();
        }
    }
    String result("unknown");
    return result;
```

```
}

int SensorService::getSensorType(int handle) const {
    size_t count = mUserSensorList.size();
    for (size_t i=0 ; i<count ; i++) {
        const Sensor& sensor(mUserSensorList[i]);
        if (sensor.getHandle() == handle) {
            return sensor.getType();
        }
    }
    return -1;
}


Vector<Sensor> SensorService::getSensorList()
{
    char value[PROPERTY_VALUE_MAX];
    property_get("debug.sensors", value, "0");
    if (atoi(value)) {
        return mUserSensorListDebug;
    }
    return mUserSensorList;
}

sp<ISensorEventConnection> SensorService::createSensorEventConnection()
{
    uid_t uid = IPCThreadState::self()->getCallingUid();
    sp<SensorEventConnection> result(new SensorEventConnection(this, uid));
    return result;
}

void SensorService::cleanupConnection(SensorEventConnection* c)
{
    Mutex::Autolock _l(mLock);
    const wp<SensorEventConnection> connection(c);
    size_t size = mActiveSensors.size();
    ALOGD_IF(DEBUG_CONNECTIONS, "%d active sensors", size);
    for (size_t i=0 ; i<size ; ) {
        int handle = mActiveSensors.keyAt(i);
        if (c->hasSensor(handle)) {
            ALOGD_IF(DEBUG_CONNECTIONS, "%i: disabling handle=0x%08x", i, handle);
            SensorInterface* sensor = mSensorMap.valueFor( handle );
            ALOGE_IF(!sensor, "mSensorMap[handle=0x%08x] is null!", handle);
            if (sensor) {
                sensor->activate(c, false);
            }
        }
        SensorRecord* rec = mActiveSensors.valueAt(i);
        ALOGE_IF(!rec, "mActiveSensors[%d] is null (handle=0x%08x)!", i, handle);
        ALOGD_IF(DEBUG_CONNECTIONS,
```

```
                    "removing connection %p for sensor[%d].handle=0x%08x",
                    c, i, handle);

            if (rec && rec->removeConnection(connection)) {
                ALOGD_IF(DEBUG_CONNECTIONS, "... and it was the last connection");
                mActiveSensors.removeItemsAt(i, 1);
                mActiveVirtualSensors.removeItem(handle);
                delete rec;
                size--;
            } else {
                i++;
            }
        }
        mActiveConnections.remove(connection);
        BatteryService::cleanup(c->getUid());
}

status_t SensorService::enable(const sp<SensorEventConnection>& connection,
            int handle)
{
        if (mInitCheck != NO_ERROR)
            return mInitCheck;

        Mutex::Autolock _l(mLock);
        SensorInterface* sensor = mSensorMap.valueFor(handle);
        SensorRecord* rec = mActiveSensors.valueFor(handle);
        if (rec == 0) {
            rec = new SensorRecord(connection);
            mActiveSensors.add(handle, rec);
            if (sensor->isVirtual()) {
                mActiveVirtualSensors.add(handle, sensor);
            }
        } else {
            if (rec->addConnection(connection)) {
                if (sensor->getSensor().getMinDelay() == 0) {
                    sensors_event_t scratch;
                    sensors_event_t& event(mLastEventSeen.editValueFor(handle));
                    if (event.version == sizeof(sensors_event_t)) {
                        connection->sendEvents(&event, 1);
                    }
                }
            }
        }

        if (connection->addSensor(handle)) {
            BatteryService::enableSensor(connection->getUid(), handle);
            if (mActiveConnections.indexOf(connection) < 0) {
                mActiveConnections.add(connection);
            }
        } else {
```

```
        ALOGW("sensor %08x already enabled in connection %p (ignoring)", handle, connection.get());
    }


    status_t err = sensor ? sensor->activate(connection.get(), true) : status_t(BAD_VALUE);

    if (err != NO_ERROR) {
        status_t resetErr = sensor ? sensor->resetStateWithoutActuatingHardware(connection.get(),
                handle) : status_t(BAD_VALUE);
        cleanupWithoutDisable(connection, handle);
    }
    return err;
}

status_t SensorService::disable(const sp<SensorEventConnection>& connection, int handle)
{
    if (mInitCheck != NO_ERROR)
        return mInitCheck;

    status_t err = cleanupWithoutDisable(connection, handle);
    if (err == NO_ERROR) {
        SensorInterface* sensor = mSensorMap.valueFor(handle);
        err = sensor ? sensor->activate(connection.get(), false) : status_t(BAD_VALUE);
    }
    return err;
}

status_t SensorService::cleanupWithoutDisable(const sp<SensorEventConnection>& connection,
        int handle) {
    Mutex::Autolock _l(mLock);
    SensorRecord* rec = mActiveSensors.valueFor(handle);
    if (rec) {
        if (connection->removeSensor(handle)) {
            BatteryService::disableSensor(connection->getUid(), handle);
        }
        if (connection->hasAnySensor() == false) {
            mActiveConnections.remove(connection);
        }
        if (rec->removeConnection(connection)) {
            mActiveSensors.removeItem(handle);
            mActiveVirtualSensors.removeItem(handle);
            delete rec;
        }
        return NO_ERROR;
    }
    return BAD_VALUE;
}

status_t SensorService::setEventRate(const sp<SensorEventConnection>& connection,
        int handle, nsecs_t ns)
{
```

```
    if (mInitCheck != NO_ERROR)
        return mInitCheck;

    SensorInterface* sensor = mSensorMap.valueFor(handle);
    if (!sensor)
        return BAD_VALUE;

    if (ns < 0)
        return BAD_VALUE;

    nsecs_t minDelayNs = sensor->getSensor().getMinDelayNs();
    if (ns < minDelayNs) {
        ns = minDelayNs;
    }

    if (ns < MINIMUM_EVENTS_PERIOD)
        ns = MINIMUM_EVENTS_PERIOD;

    return sensor->setDelay(connection.get(), handle, ns);
}

// ---------------------------------------------------------------------------

SensorService::SensorRecord::SensorRecord(
        const sp<SensorEventConnection>& connection)
{
    mConnections.add(connection);
}

bool SensorService::SensorRecord::addConnection(
        const sp<SensorEventConnection>& connection)
{
    if (mConnections.indexOf(connection) < 0) {
        mConnections.add(connection);
        return true;
    }
    return false;
}

bool SensorService::SensorRecord::removeConnection(
        const wp<SensorEventConnection>& connection)
{
    ssize_t index = mConnections.indexOf(connection);
    if (index >= 0) {
        mConnections.removeItemsAt(index, 1);
    }
    return mConnections.size() ? false : true;
}

// ---------------------------------------------------------------------------
```

```
SensorService::SensorEventConnection::SensorEventConnection(
        const sp<SensorService>& service, uid_t uid)
    : mService(service), mChannel(new BitTube()), mUid(uid)
{
}

SensorService::SensorEventConnection::~SensorEventConnection()
{
    ALOGD_IF(DEBUG_CONNECTIONS, "~SensorEventConnection(%p)", this);
    mService->cleanupConnection(this);
}

void SensorService::SensorEventConnection::onFirstRef()
{
}

bool SensorService::SensorEventConnection::addSensor(int32_t handle) {
    Mutex::Autolock _l(mConnectionLock);
    if (mSensorInfo.indexOf(handle) < 0) {
        mSensorInfo.add(handle);
        return true;
    }
    return false;
}

bool SensorService::SensorEventConnection::removeSensor(int32_t handle) {
    Mutex::Autolock _l(mConnectionLock);
    if (mSensorInfo.remove(handle) >= 0) {
        return true;
    }
    return false;
}

bool SensorService::SensorEventConnection::hasSensor(int32_t handle) const {
    Mutex::Autolock _l(mConnectionLock);
    return mSensorInfo.indexOf(handle) >= 0;
}

bool SensorService::SensorEventConnection::hasAnySensor() const {
    Mutex::Autolock _l(mConnectionLock);
    return mSensorInfo.size() ? true : false;
}

status_t SensorService::SensorEventConnection::sendEvents(
        sensors_event_t const* buffer, size_t numEvents,
        sensors_event_t* scratch)
{
    size_t count = 0;
    if (scratch) {
        Mutex::Autolock _l(mConnectionLock);
        size_t i=0;
```

```
        while (i<numEvents) {
            const int32_t curr = buffer[i].sensor;
            if (mSensorInfo.indexOf(curr) >= 0) {
                do {
                    scratch[count++] = buffer[i++];
                } while ((i<numEvents) && (buffer[i].sensor == curr));
            } else {
                i++;
            }
        }
    } else {
        scratch = const_cast<sensors_event_t *>(buffer);
        count = numEvents;
    }

    ssize_t size = SensorEventQueue::write(mChannel,
            reinterpret_cast<ASensorEvent const*>(scratch), count);
    if (size == -EAGAIN) {
        return size;
    }

    return size < 0 ? status_t(size) : status_t(NO_ERROR);
}

sp<BitTube> SensorService::SensorEventConnection::getSensorChannel() const
{
    return mChannel;
}

status_t SensorService::SensorEventConnection::enableDisable(int handle, bool enabled)
{
    status_t err;
    if (enabled) {
        err = mService->enable(this, handle);
    } else {
        err = mService->disable(this, handle);
    }
    return err;
}

status_t SensorService::SensorEventConnection::setEventRate(int handle, nsecs_t ns)
{
    return mService->setEventRate(this, handle, ns);
}

// ---------------------------------------------------------------------------
};
```

通过上述实现代码可以了解 SensorService 服务的创建、启动过程，整个过程的 C/S 通信架构如图 11-5 所示。

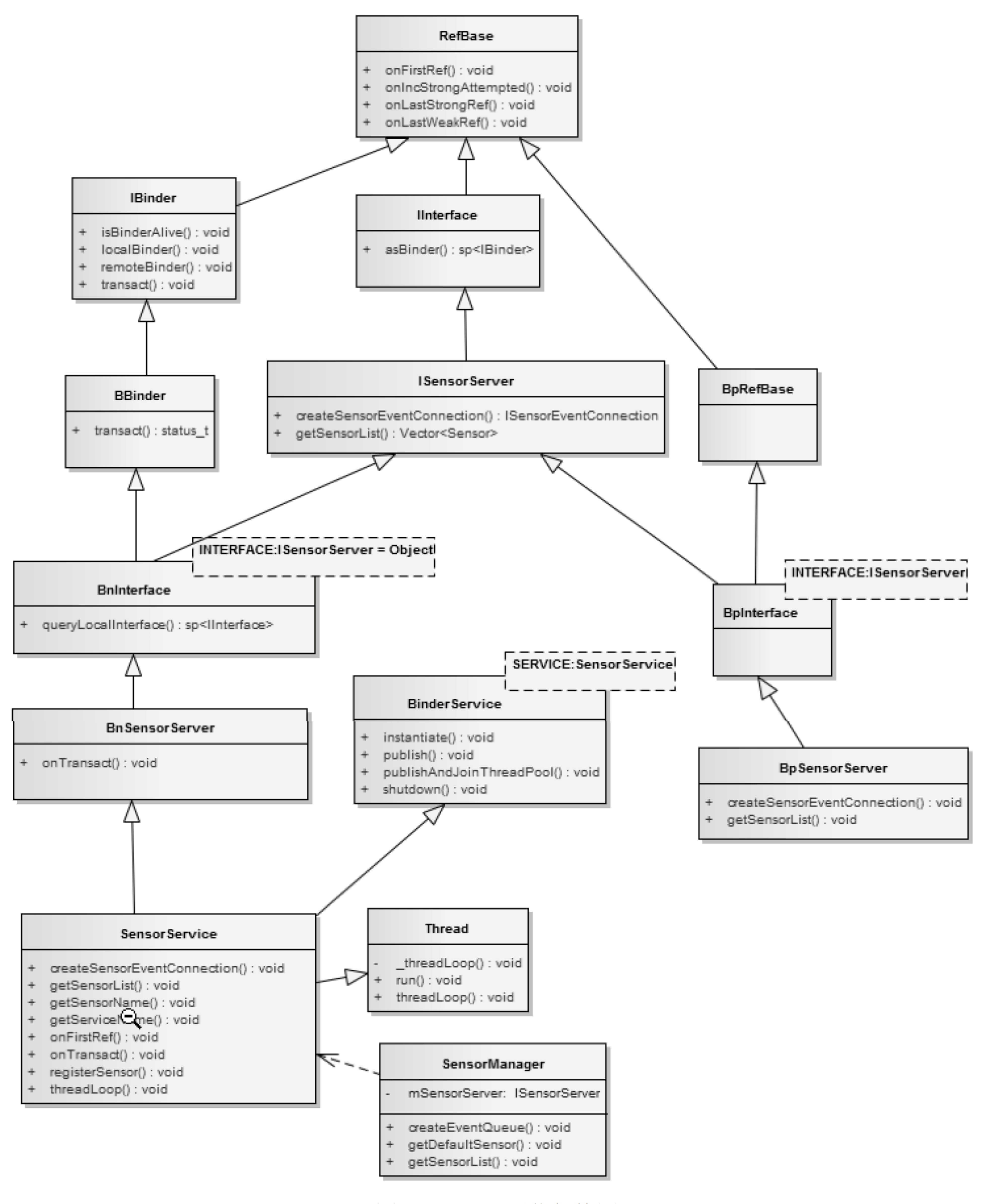


图 11-5　C/S 通信架构图

在此需要注意，BpSensorServer 并没有在系统中被用到，即使从 ISensorServer.cpp 中把它删除也不会对 Sensor 的工作有任何影响。这是因为它的工作已经被 SensorManager.cpp 所取代，ServiceManager 会直接获取上面 System_init 文件中添加的 SensorService 对象。

11.4.4　封装 HAL 层的代码

在 Android 系统中，通过文件 frameworks/native/services/sensorservice/SensorDevice.cpp 封装了 HAL 层的代码，主要包含的功能如下所示。

☑ 获取 Sensor 列表（getSensorList）。
☑ 获取 Sensor 事件（poll）。
☑ Enable 或 Disable sensor（activate）。
☑ 设置 delay 时间。

文件 SensorDevice.cpp 的具体实现代码如下所示。

```
namespace android {
// ---------------------------------------------------------------------------

ANDROID_SINGLETON_STATIC_INSTANCE(SensorDevice)

SensorDevice::SensorDevice()
    :  mSensorDevice(0),
       mSensorModule(0)
{
    status_t err = hw_get_module(SENSORS_HARDWARE_MODULE_ID,
            (hw_module_t const**)&mSensorModule);

    ALOGE_IF(err, "couldn't load %s module (%s)",
            SENSORS_HARDWARE_MODULE_ID, strerror(-err));

    if (mSensorModule) {
        err = sensors_open(&mSensorModule->common, &mSensorDevice);

        ALOGE_IF(err, "couldn't open device for module %s (%s)",
                SENSORS_HARDWARE_MODULE_ID, strerror(-err));

        if (mSensorDevice) {
            sensor_t const* list;
            ssize_t count = mSensorModule->get_sensors_list(mSensorModule, &list);
            mActivationCount.setCapacity(count);
            Info model;
            for (size_t i=0 ; i<size_t(count) ; i++) {
                mActivationCount.add(list[i].handle, model);
                mSensorDevice->activate(mSensorDevice, list[i].handle, 0);
            }
        }
    }
}

void SensorDevice::dump(String8& result, char* buffer, size_t SIZE)
{
    if (!mSensorModule) return;
    sensor_t const* list;
    ssize_t count = mSensorModule->get_sensors_list(mSensorModule, &list);

    snprintf(buffer, SIZE, "%d h/w sensors:\n", int(count));
    result.append(buffer);

    Mutex::Autolock _l(mLock);
```

```
    for (size_t i=0 ; i<size_t(count) ; i++) {
        const Info& info = mActivationCount.valueFor(list[i].handle);
        snprintf(buffer, SIZE, "handle=0x%08x, active-count=%d, rates(ms)={ ",
                list[i].handle,
                info.rates.size());
        result.append(buffer);
        for (size_t j=0 ; j<info.rates.size() ; j++) {
            snprintf(buffer, SIZE, "%4.1f%s",
                    info.rates.valueAt(j) / 1e6f,
                    j<info.rates.size()-1 ? ", " : "");
            result.append(buffer);
        }
        snprintf(buffer, SIZE, " }, selected=%4.1f ms\n",    info.delay / 1e6f);
        result.append(buffer);
    }
}

ssize_t SensorDevice::getSensorList(sensor_t const** list) {
    if (!mSensorModule) return NO_INIT;
    ssize_t count = mSensorModule->get_sensors_list(mSensorModule, list);
    return count;
}

status_t SensorDevice::initCheck() const {
    return mSensorDevice && mSensorModule ? NO_ERROR : NO_INIT;
}

ssize_t SensorDevice::poll(sensors_event_t* buffer, size_t count) {
    if (!mSensorDevice) return NO_INIT;
    ssize_t c;
    do {
        c = mSensorDevice->poll(mSensorDevice, buffer, count);
    } while (c == -EINTR);
    return c;
}

status_t SensorDevice::resetStateWithoutActuatingHardware(void *ident, int handle)
{
    if (!mSensorDevice) return NO_INIT;
    Info& info( mActivationCount.editValueFor(handle));
    Mutex::Autolock _l(mLock);
    info.rates.removeItem(ident);
    return NO_ERROR;
}

status_t SensorDevice::activate(void* ident, int handle, int enabled)
{
    if (!mSensorDevice) return NO_INIT;
    status_t err(NO_ERROR);
    bool actuateHardware = false;
```

```
    Info& info( mActivationCount.editValueFor(handle) );


    ALOGD_IF(DEBUG_CONNECTIONS,
            "SensorDevice::activate: ident=%p, handle=0x%08x, enabled=%d, count=%d",
            ident, handle, enabled, info.rates.size());

    if (enabled) {
        Mutex::Autolock _l(mLock);
        ALOGD_IF(DEBUG_CONNECTIONS, "... index=%ld",
                info.rates.indexOfKey(ident));

        if (info.rates.indexOfKey(ident) < 0) {
            info.rates.add(ident, DEFAULT_EVENTS_PERIOD);
            if (info.rates.size() == 1) {
                actuateHardware = true;
            }
        } else {
        }
    } else {
        Mutex::Autolock _l(mLock);
        ALOGD_IF(DEBUG_CONNECTIONS, "... index=%ld",
                info.rates.indexOfKey(ident));

        ssize_t idx = info.rates.removeItem(ident);
        if (idx >= 0) {
            if (info.rates.size() == 0) {
                actuateHardware = true;
            }
        } else {
        }
    }

    if (actuateHardware) {
        ALOGD_IF(DEBUG_CONNECTIONS, "\t>>> actuating h/w");

        err = mSensorDevice->activate(mSensorDevice, handle, enabled);
        ALOGE_IF(err, "Error %s sensor %d (%s)",
                enabled ? "activating" : "disabling",
                handle, strerror(-err));
    }

    {
        Mutex::Autolock _l(mLock);
        nsecs_t ns = info.selectDelay();
        mSensorDevice->setDelay(mSensorDevice, handle, ns);
    }

    return err;
}
```

```
status_t SensorDevice::setDelay(void* ident, int handle, int64_t ns)
{
    if (!mSensorDevice) return NO_INIT;
    Mutex::Autolock _l(mLock);
    Info& info( mActivationCount.editValueFor(handle) );
    status_t err = info.setDelayForIdent(ident, ns);
    if (err < 0) return err;
    ns = info.selectDelay();
    return mSensorDevice->setDelay(mSensorDevice, handle, ns);
}

int SensorDevice::getHalDeviceVersion() const {
    if (!mSensorDevice) return -1;

    return mSensorDevice->common.version;
}

// ---------------------------------------------------------------------------

status_t SensorDevice::Info::setDelayForIdent(void* ident, int64_t ns)
{
    ssize_t index = rates.indexOfKey(ident);
    if (index < 0) {
        ALOGE("Info::setDelayForIdent(ident=%p, ns=%lld) failed (%s)",
                ident, ns, strerror(-index));
        return BAD_INDEX;
    }
    rates.editValueAt(index) = ns;
    return NO_ERROR;
}

nsecs_t SensorDevice::Info::selectDelay()
{
    nsecs_t ns = rates.valueAt(0);
    for (size_t i=1 ; i<rates.size() ; i++) {
        nsecs_t cur = rates.valueAt(i);
        if (cur < ns) {
            ns = cur;
        }
    }
    delay = ns;
    return ns;
}

// ---------------------------------------------------------------------------
};
```

这样 SensorService 会把任务交给 SensorDevice，而 SensorDevice 会调用标准的抽象层接口。由此可见，Sensor 架构的抽象层接口是最标准的一种，它很好地实现了抽象层与本地框架的分离。

11.4.5 处理消息队列

在 Android 的传感器系统中，文件 frameworks/native/libs/gui/SensorEventQueue.cpp 实现了处理消息队列的功能。此文件能够在创建其实例时传入 SensorEventConnection 的实例，SensorEventConnection 继承于 ISensorEventConnection。SensorEventConnection 其实是客户端调用 SensorService 的 createSensorEventConnection()方法创建的，是客户端与服务端沟通的桥梁，通过这个桥梁，可以完成如下任务。

☑ 获取管道的句柄。
☑ 往管道读写数据。
☑ 通知服务端对 Sensor 使能。

文件 frameworks/native/libs/gui/SensorEventQueue.cpp 的具体实现代码如下所示。

```
// ----------------------------------------------------------------------------
namespace android {
// ----------------------------------------------------------------------------

SensorEventQueue::SensorEventQueue(const sp<ISensorEventConnection>& connection)
    : mSensorEventConnection(connection)
{
}

SensorEventQueue::~SensorEventQueue()
{
}

void SensorEventQueue::onFirstRef()
{
    mSensorChannel = mSensorEventConnection->getSensorChannel();
}

int SensorEventQueue::getFd() const
{
    return mSensorChannel->getFd();
}


ssize_t SensorEventQueue::write(const sp<BitTube>& tube,
        ASensorEvent const* events, size_t numEvents) {
    return BitTube::sendObjects(tube, events, numEvents);
}

ssize_t SensorEventQueue::read(ASensorEvent* events, size_t numEvents)
{
    return BitTube::recvObjects(mSensorChannel, events, numEvents);
}
```

```
sp<Looper> SensorEventQueue::getLooper() const
{
    Mutex::Autolock _l(mLock);
    if (mLooper == 0) {
        mLooper = new Looper(true);
        mLooper->addFd(getFd(), getFd(), ALOOPER_EVENT_INPUT, NULL, NULL);
    }
    return mLooper;
}

status_t SensorEventQueue::waitForEvent() const
{
    const int fd = getFd();
    sp<Looper> looper(getLooper());

    int events;
    int32_t result;
    do {
        result = looper->pollOnce(-1, NULL, &events, NULL);
        if (result == ALOOPER_POLL_ERROR) {
            ALOGE("SensorEventQueue::waitForEvent error (errno=%d)", errno);
            result = -EPIPE;
            break;
        }
        if (events & ALOOPER_EVENT_HANGUP) {
            ALOGE("SensorEventQueue::waitForEvent error HANGUP");
            result = -EPIPE;
            break;
        }
    } while (result != fd);

    return   (result == fd) ? status_t(NO_ERROR) : result;
}

status_t SensorEventQueue::wake() const
{
    sp<Looper> looper(getLooper());
    looper->wake();
    return NO_ERROR;
}

status_t SensorEventQueue::enableSensor(Sensor const* sensor) const {
    return mSensorEventConnection->enableDisable(sensor->getHandle(), true);
}

status_t SensorEventQueue::disableSensor(Sensor const* sensor) const {
    return mSensorEventConnection->enableDisable(sensor->getHandle(), false);
}
```

```
status_t SensorEventQueue::enableSensor(int32_t handle, int32_t us) const {
    status_t err = mSensorEventConnection->enableDisable(handle, true);
    if (err == NO_ERROR) {
        mSensorEventConnection->setEventRate(handle, us2ns(us));
    }
    return err;
}

status_t SensorEventQueue::disableSensor(int32_t handle) const {
    return mSensorEventConnection->enableDisable(handle, false);
}

status_t SensorEventQueue::setEventRate(Sensor const* sensor, nsecs_t ns) const {
    return mSensorEventConnection->setEventRate(sensor->getHandle(), ns);
}

// ---------------------------------------------------------------------------
};
```

由此可见，SensorManager 负责控制流，通过 C/S 的 Binder 机制与 SensorService 实现通信。具体过程如图 11-6 所示。

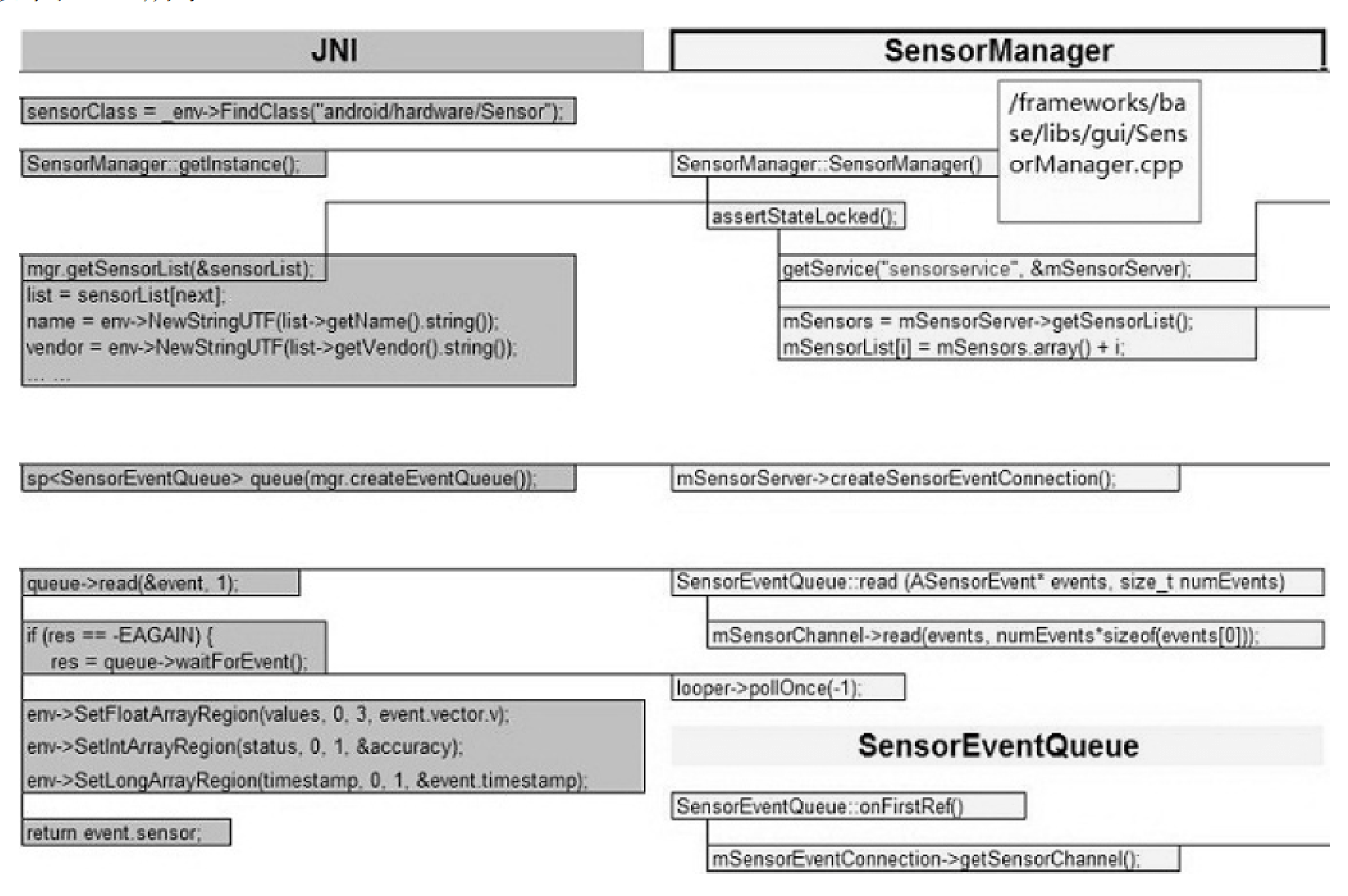


图 11-6 SensorManager 控制流的处理流程

而 SensorEventQueue 负责数据流，功能是通过管道机制来读写底层的数据。具体过程如图 11-7 所示。



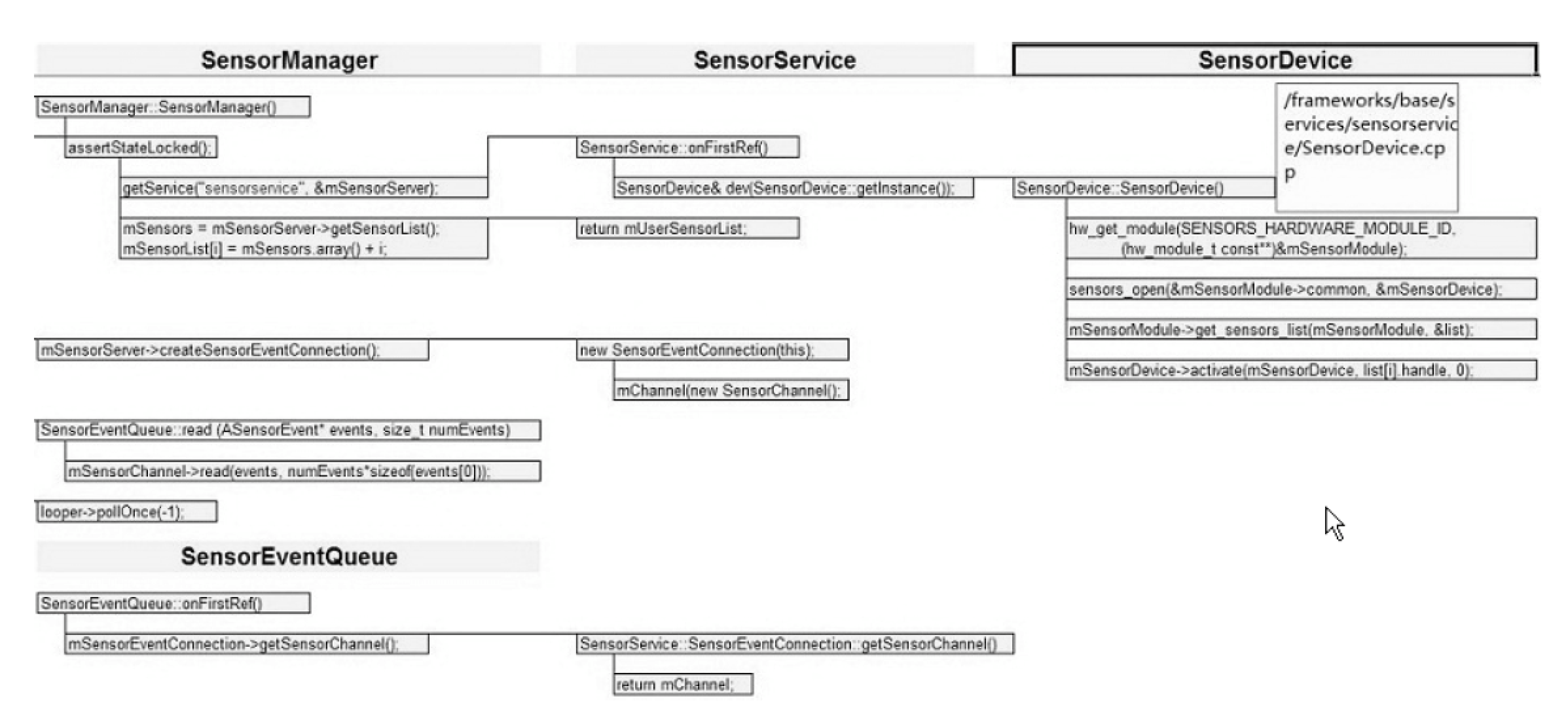


图 11-7　SensorEventQueue 数据流的处理流程

11.5　HAL 层详解

在 Android 系统中，在 HAL 层中提供了 Android 独立于具体硬件的抽象接口。其中 HAL 层的头文件路径是 hardware/libhardware/include/hardware/sensors.h。

而具体实现文件需要开发者个人编写，可以参考文件 Hardware/invensense/libsensors_iio/sensors_mpl.cpp。

文件 sensors.h 的主要实现代码如下所示。

```
typedef struct {
    union {
        float v[3];
        struct {
            float x;
            float y;
            float z;
        };
        struct {
            float azimuth;
            float pitch;
            float roll;
        };
    };
    int8_t status;
    uint8_t reserved[3];
} sensors_vec_t;

typedef struct {
  union {
```

```
        float uncalib[3];
        struct {
            float x_uncalib;
            float y_uncalib;
            float z_uncalib;
        };
    };
    union {
        float bias[3];
        struct {
            float x_bias;
            float y_bias;
            float z_bias;
        };
    };
} uncalibrated_event_t;

/**
 * Union of the various types of sensor data
 * that can be returned
 */
typedef struct sensors_event_t {
        /* must be sizeof(struct sensors_event_t) */
        int32_t version;

        /* sensor identifier */
        int32_t sensor;

        /* sensor type */
        int32_t type;

        /* reserved */
        int32_t reserved0;

        /* time is in nanosecond */
        int64_t timestamp;

        union {
                float data[16];

                /* acceleration values are in meter per second per second (m/s^2) */
                sensors_vec_t acceleration;

                /* magnetic vector values are in micro-Tesla (uT) */
                sensors_vec_t magnetic;

                /* orientation values are in degrees */
                sensors_vec_t orientation;

                /* gyroscope values are in rad/s */
                sensors_vec_t gyro;
```

```
            /* temperature is in degrees centigrade (Celsius) */
            float temperature;

            /* distance in centimeters */
            float distance;

            /* light in SI lux units */
            float light;

            /* pressure in hectopascal (hPa) */
            float pressure;

            /* relative humidity in percent */
            float relative_humidity;

            /* step-counter */
            uint64_t step_counter;

            /* uncalibrated gyroscope values are in rad/s */
            uncalibrated_event_t uncalibrated_gyro;

            /* uncalibrated magnetometer values are in micro-Teslas */
            uncalibrated_event_t uncalibrated_magnetic;
        };
        uint32_t reserved1[4];
} sensors_event_t;

struct sensor_t;
struct sensors_module_t {
        struct hw_module_t common;
        int (*get_sensors_list)(struct sensors_module_t* module,
                        struct sensor_t const** list);
};

struct sensor_t {

        /* Name of this sensor.
         * All sensors of the same "type" must have a different "name"
         */
        const char* name;

        /* vendor of the hardware part */
        const char* vendor;
        int version;
        int handle;

        /* this sensor's type. */
        int type;

        /* maximum range of this sensor's value in SI units */
```

```
    float maxRange;

    /* smallest difference between two values reported by this sensor */
    float resolution;

    /*  传感器的粗略值，单位毫安*/
    float power;
    int32_t minDelay;

    /*保留字段，必须为 0*/
    void* reserved[8];
};

struct sensors_poll_device_t {
    struct hw_device_t common;
    int (*activate)(struct sensors_poll_device_t *dev,
            int handle, int enabled);
    int (*setDelay)(struct sensors_poll_device_t *dev,
            int handle, int64_t ns);
    int (*poll)(struct sensors_poll_device_t *dev,
            sensors_event_t* data, int count);
};
typedef struct sensors_poll_device_1 {
    union {
        struct sensors_poll_device_t v0;

        struct {
            struct hw_device_t common;
            int (*activate)(struct sensors_poll_device_t *dev,
                    int handle, int enabled);

            int (*setDelay)(struct sensors_poll_device_t *dev,
                    int handle, int64_t period_ns);

            int (*poll)(struct sensors_poll_device_t *dev,
                    sensors_event_t* data, int count);
        };
    };
    int (*batch)(struct sensors_poll_device_1* dev,
            int handle, int flags, int64_t period_ns, int64_t timeout);

    void (*reserved_procs[8])(void);

} sensors_poll_device_1_t;



/**用于方便打开和关闭装置的 API */

static inline int sensors_open(const struct hw_module_t* module,
        struct sensors_poll_device_t** device) {
```

```
    return module->methods->open(module,
            SENSORS_HARDWARE_POLL, (struct hw_device_t**)device);
}

static inline int sensors_close(struct sensors_poll_device_t* device) {
    return device->common.close(&device->common);
}

static inline int sensors_open_1(const struct hw_module_t* module,
        sensors_poll_device_1_t** device) {
    return module->methods->open(module,
            SENSORS_HARDWARE_POLL, (struct hw_device_t**)device);
}

static inline int sensors_close_1(sensors_poll_device_1_t* device) {
    return device->common.close(&device->common);
}

__END_DECLS

#endif
```

而具体的实现文件是 Linux Kernel 层，也就是具体的硬件设备驱动程序，例如，可以将其命名为 sensors.c，然后编写如下定义 struct sensors_poll_device_t 的代码。

```
struct sensors_poll_device_t {
    struct hw_device_t common;

    int (*activate)(struct sensors_poll_device_t *dev,
            int handle, int enabled);

    //对于一个给定的传感器，设置在事件之间的延迟
    int (*setDelay)(struct sensors_poll_device_t *dev,
            int handle, int64_t ns);

    //返回传感器数据的数组
    int (*poll)(struct sensors_poll_device_t *dev,
            sensors_event_t* data, int count);
};
```

可以编写如下定义 struct sensors_module_t 的代码。

```
struct sensors_module_t {
    struct hw_module_t common;

    /**
     *枚举所有可用的传感器
     * @return number of sensors in the list
     */
    int (*get_sensors_list)(struct sensors_module_t* module,
            struct sensor_t const** list);
};
```

可以编写如下定义 struct sensor_t 的代码。

```
struct sensor_t {
    /*传感器名字*/
    const char* name;
    /*硬件部分的供应商*/
    const char* vendor;
    /*版本硬件驱动*/
    int version;
    /*处理标识此传感器。此句柄用于激活和关闭该传感器。在这个版本的 API 的手柄值必须是 8 位的
    */
    int handle;
    /*传感器类型*/
    int type;
    /*该传感器的 maximaum 范围值*/
    float maxRange;
    /* smallest difference between two values reported by this sensor */
    float resolution;
    /* rough estimate of this sensor's power consumption in mA */
    float power;
    /* minimum delay allowed between events in microseconds. A value of zero
     * means that this sensor doesn't report events at a constant rate, but
     * rather only when a new data is available */
    int32_t minDelay;
    /* reserved fields, must be zero */
    void* reserved[8];
};
```

可以编写如下定义 struct sensors_event_t 的代码。

```
typedef struct {
    union {
        float v[3];
        struct {
            float x;
            float y;
            float z;
        };
        struct {
            float azimuth;
            float pitch;
            float roll;
        };
    };
    int8_t status;
    uint8_t reserved[3];
} sensors_vec_t;

/**
 *各种类型的传感器数据可以被返回的联合类型
```

```
 */
typedef struct sensors_event_t {
    /*必须是 struct sensors_event_t 类型*/
    int32_t version;

    /*传感器标识*/
    int32_t sensor;

    /*传感器类型*/
    int32_t type;

    /*保留的*/
    int32_t reserved0;

    /*纳秒时间*/
    int64_t timestamp;

    union {
        float data[16];

        /*加速度值，单位是 m/s2*/
        sensors_vec_t acceleration;

        /*磁矢量值，微特斯拉（uT）*/
        sensors_vec_t magnetic;

        /*某一度的定向值*/
        sensors_vec_t orientation;

        /*陀螺仪值，单位为 rad/s*/
        sensors_vec_t gyro;

        /*温度，单位是摄氏度*/
        float temperature;

        /*距离，单位是厘米*/
        float distance;

        /*光亮度*/
        float light;

        /*压力，单位是 hPa*/
        float pressure;

        /*相对湿度*/
        float relative_humidity;
    };
    uint32_t reserved1[4];
} sensors_event_t;
```

可以编写如下定义 struct sensors_module_t 的代码。

```
static const struct sensor_t sSensorList[] = {
        { "MMA8452Q 3-axis Accelerometer",
                "Freescale Semiconductor",
                1, SENSORS_HANDLE_BASE+ID_A,
                SENSOR_TYPE_ACCELEROMETER, 4.0f*9.81f, (4.0f*9.81f)/256.0f, 0.2f, 0, { } },
        { "AK8975 3-axis Magnetic field sensor",
                "Asahi Kasei",
                1, SENSORS_HANDLE_BASE+ID_M,
                SENSOR_TYPE_MAGNETIC_FIELD, 2000.0f, 1.0f/16.0f, 6.8f, 0, { } },
        { "AK8975 Orientation sensor",
                "Asahi Kasei",
                1, SENSORS_HANDLE_BASE+ID_O,
                SENSOR_TYPE_ORIENTATION, 360.0f, 1.0f, 7.0f, 0, { } },

    { "ST 3-axis Gyroscope sensor",
          "STMicroelectronics",
          1, SENSORS_HANDLE_BASE+ID_GY,
          SENSOR_TYPE_GYROSCOPE, RANGE_GYRO, CONVERT_GYRO, 6.1f, 1190, { } },

    { "AL3006Proximity sensor",
        "Dyna Image Corporation",
        1, SENSORS_HANDLE_BASE+ID_P,
        SENSOR_TYPE_PROXIMITY,
        PROXIMITY_THRESHOLD_CM, PROXIMITY_THRESHOLD_CM,
        0.5f, 0, { } },

        { "AL3006 light sensor",
                "Dyna Image Corporation",
                1, SENSORS_HANDLE_BASE+ID_L,
                SENSOR_TYPE_LIGHT, 10240.0f, 1.0f, 0.5f, 0, { } },

};

static int open_sensors(const struct hw_module_t* module, const char* name,
        struct hw_device_t** device);

static int sensors__get_sensors_list(struct sensors_module_t* module,
        struct sensor_t const** list)
{
    *list = sSensorList;
    return ARRAY_SIZE(sSensorList);
}

static struct hw_module_methods_t sensors_module_methods = {
    .open = open_sensors
};
```

```
const struct sensors_module_t HAL_MODULE_INFO_SYM = {
    .common = {
        .tag = HARDWARE_MODULE_TAG,
        .version_major = 1,
        .version_minor = 0,
        .id = SENSORS_HARDWARE_MODULE_ID,
        .name = "MMA8451Q & AK8973A & gyro Sensors Module",
        .author = "The Android Project",
        .methods = &sensors_module_methods,
    },
    .get_sensors_list = sensors__get_sensors_list
};

static int open_sensors(const struct hw_module_t* module, const char* name,
        struct hw_device_t** device)
{
    return init_nusensors(module, device); //待后面讲解
}
```

到此为止，整个 Android 系统中传感器模块的源码分析完毕。由此可见，整个传感器系统的总体调用关系如图 11-8 所示。

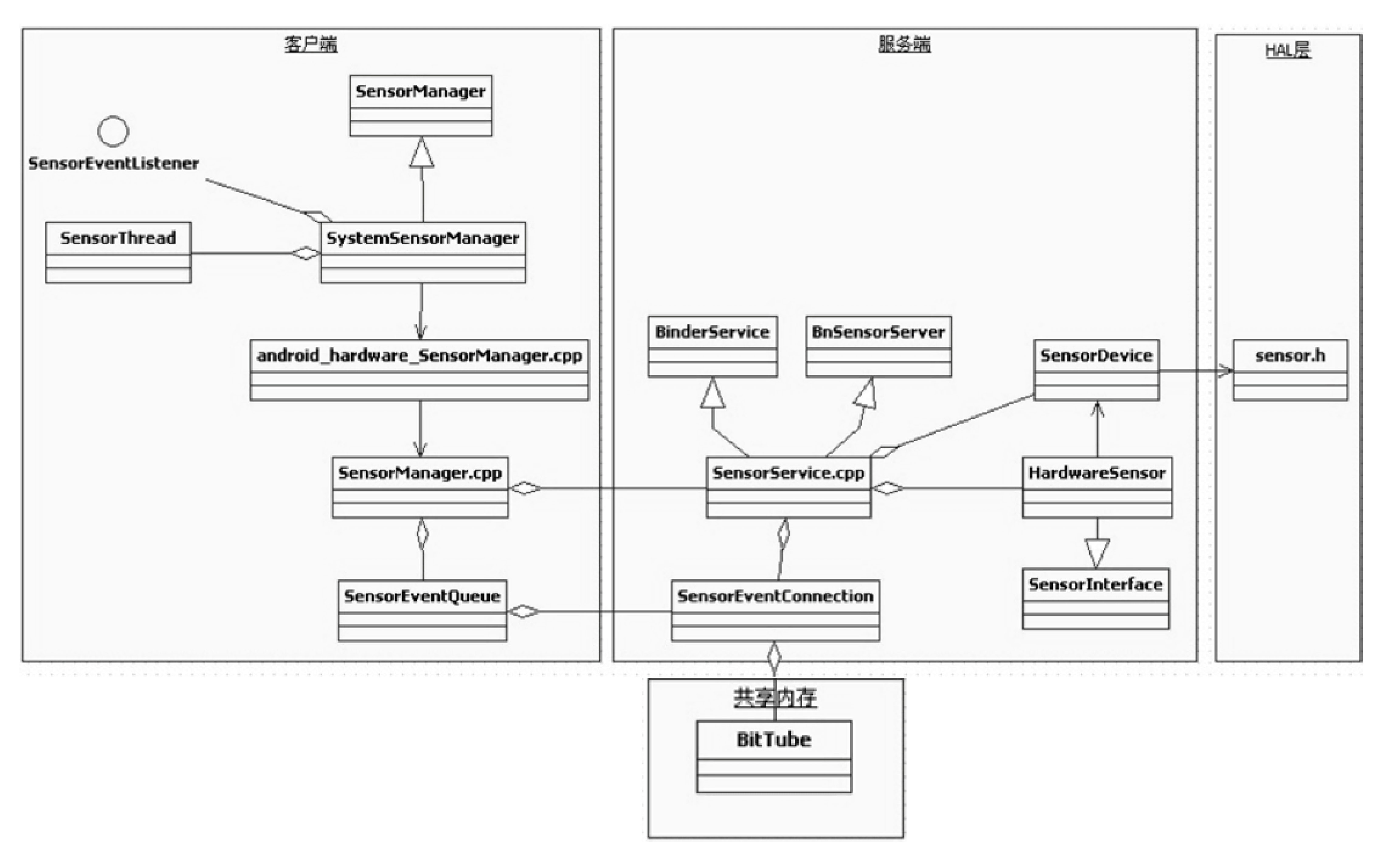


图 11-8　传感器系统的总体调用关系

客户端读取数据时的调用时序如图 11-9 所示。

服务器端的调用时序如图 11-10 所示。

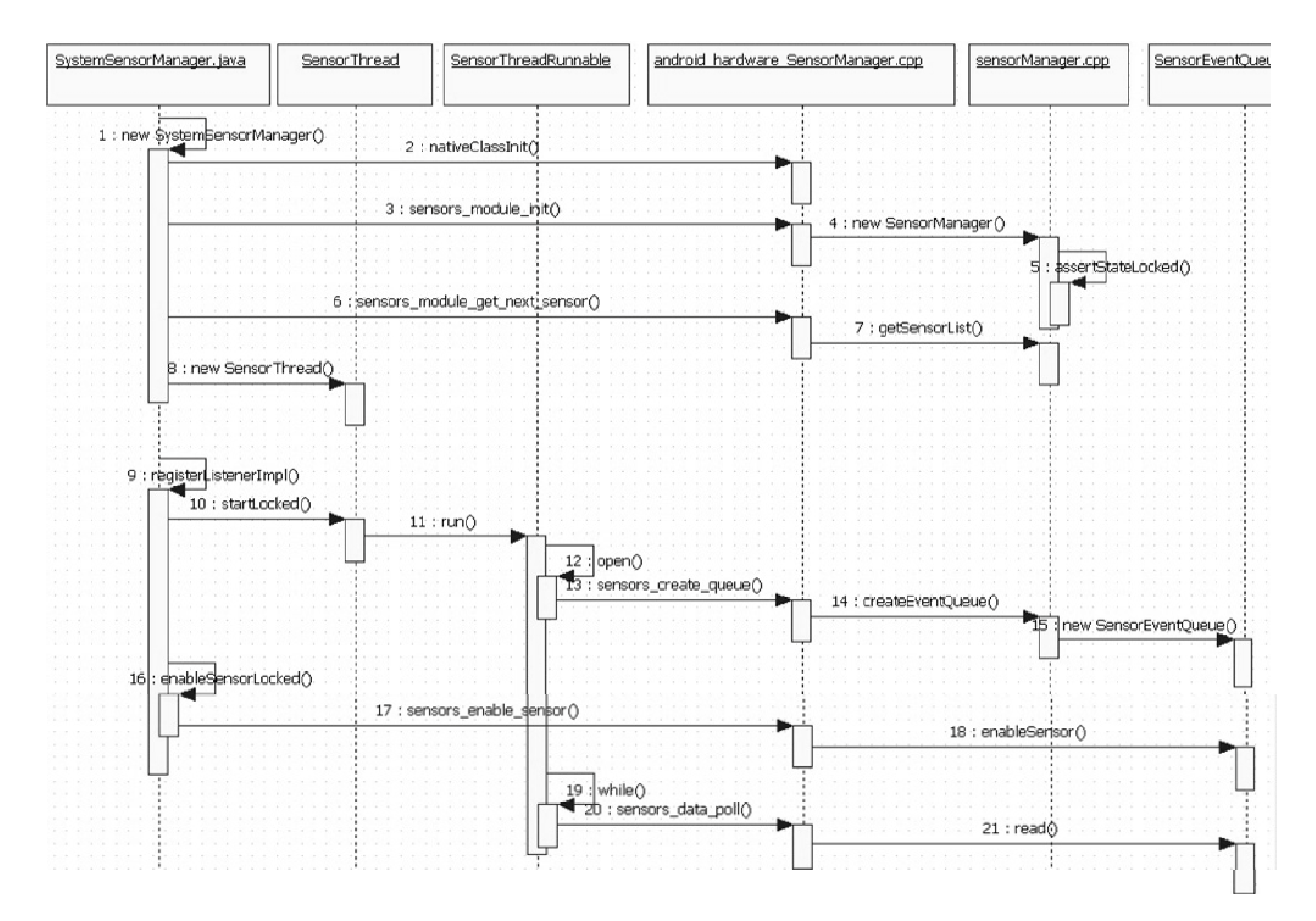


图 11-9　客户端读取数据时的调用时序图

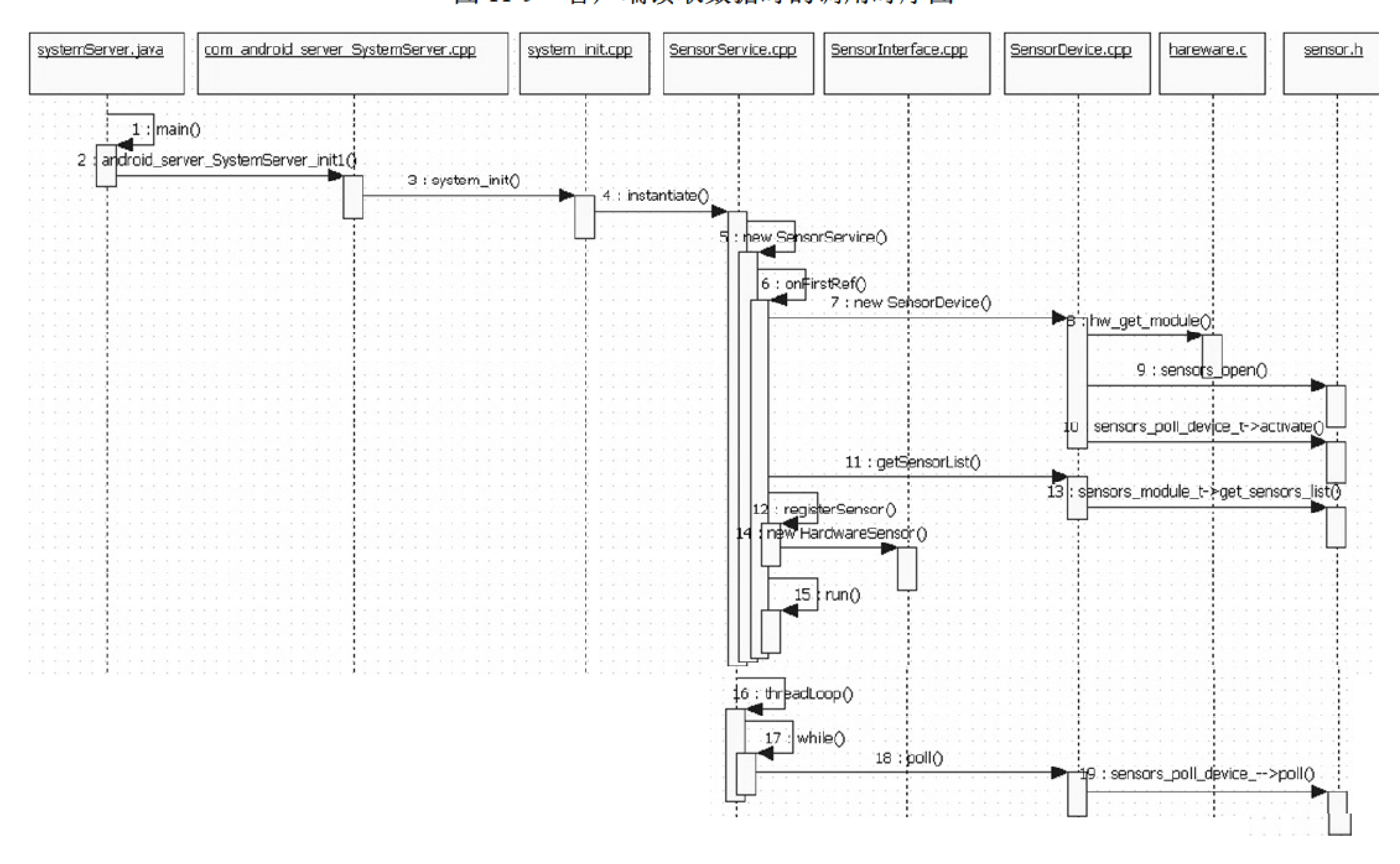


图 11-10　服务器端的调用时序图



第 12 章　蓝牙系统架构详解

蓝牙这一名称来自于第十世纪的一位丹麦国王 Harald Blatand，Blatand 在英文里的意思可以被解释为 Bluetooth。通过蓝牙技术可以有效地简化移动通信终端设备之间的通信，也能够成功地简化设备与 Internet 之间的通信，从而使数据传输变得更加迅速高效，为无线通信拓宽道路。本章将详细讲解 Android 系统中蓝牙系统的核心架构知识，为读者学习本书后面的知识打下基础。

12.1　短距离无线通信技术概览

在物联网中物与网相连的最后数米，发挥关键作用的是短距离无线传输技术。目前有多种短距离无线传输技术可以应用在物联网中，在我国，除已经得到大规模应用的 RFID 之外，还有 Wi-Fi、ZigBee、蓝牙等比较成熟的技术，以及基于这些技术发展而来的新技术。这些技术各具特点，因对其传输速度、距离、耗电量等方面的要求不同，形成了各自不同的物联网应用场景。本节将简要介绍当今实现短距离无线通信的常用技术。

12.1.1　ZigBee——低功耗、自组网

ZigBee 以其鲜明的技术特点在物联网中受到了高度关注，该技术使用的频段分别为 2.4GHz、868MHz（欧洲）及 915MHz（美国）。其主要的技术特点：一是数据传输速率低，只有 10Kbps～250Kbps；二是功耗低，低传输速率带来了仅为 1 毫瓦的低发射功率，据估算，ZigBee 设备仅靠两节 5 号电池就可以维持长达 6 个月到两年的使用时间，这是 ZigBee 的一个独特优势；三是成本低，因为 ZigBee 传输速率低、协议简单；四是网络容量大，每个 ZigBee 网络最多可以支持 255 个设备，一个区域内可以同时存在最多 100 个 ZigBee 网络，网络组成灵活。ZigBee 芯片主要企业有德州仪器、飞思卡尔等。市场调研机构 ABI Research 的一份数据显示，2005—2012 年，ZigBee 市场的年均复合增长率为 63%。

“ZigBee 是从家庭自动化开始的，在瑞典哥德堡就是从智能电表开始，然后进一步用到燃气表、水表、热力表等家庭各种计量表。”在 2011 年中国无线世界暨物联网大会上，ZigBee 联盟大中华区代表黄家瑞说，“ZigBee 在智能电表里不仅仅是远程抄表工具，它是一个终端，也是一个网关，这些网关结合在一起，整个小区就变成了智能电网小区，智能电表可以搜集家里所有家电的用电信息。”

目前，ZigBee 正在完善其网关标准，2011 年 7 月底发布了第十个标准 ZigBee Gateway（ZigBee 网关）。ZigBee Gateway 提供了一种简单、高成本效益的互联网连接方式，使服务提供商、企业和个人消费者有机会运行这些设备并将 ZigBee 网络连接至互联网。ZigBee Gateway 是 ZigBee Network Devicesp（ZigBee 网络设备）这一新类别范畴的首个标准，这将使 ZigBee 发展进一步提速。

12.1.2　Wi-Fi——大带宽支持家庭互联

Wi-Fi 是以太网的一种无线扩展技术，如果有多个用户同时通过一个热点接入，带宽将被这些用户

共享，Wi-Fi 的速率会降低，处于 2.4GHz 频段的 Wi-Fi 信号受墙壁阻隔的影响较小。Wi-Fi 的传输速率随着技术的演进还在不断提高，我国电信运营商在构建无线城市中采用的 Wi-Fi 技术部分已经升级到 802.11n，最高速率从 802.11g 标准的 11Mbps 提高到 50Mbps 以上。在 Wi-Fi 产业链中，最大的芯片企业是博通。

“在过去几年里整个 Wi-Fi 技术和产品发货量达到 20 亿个，整个 Wi-Fi 产品销售每年都是以两位数的速度持续增长。”Wi-Fi 联盟董事 Myron Hattig 说，“在 2011 年我们还会销售 10 亿个产品。”

在笔记本电脑和手机上已经得到广泛应用的 Wi-Fi 正在向消费电子产品渗透，Myron Hattig 说：“除了手机外，已经有 25%的消费类电子设备使用 Wi-Fi，在打印机、洗衣机上都在使用 Wi-Fi，家用电器生产商协会将 Wi-Fi 作为一个更高级别的智能电器沟通技术。Wi-Fi 可以将设备与设备相连，从而使整个家庭的家用电器、电子设备相连。”

最大 Wi-Fi 芯片制造商博通正在推动 Wi-Fi Direct 标准的商用，以支持这种设备到设备的直连。特别是在家庭互联中，相片、视频等大数据量的业务在手机、平板电脑、电视等设备中的直连应用前景广阔。Myron Hattig 告诉记者：“直接技术可将平板电脑的内容展示在电视上，相关产品会在 2012 年发布。”

基于 Wi-Fi 上发展起来的 WIGIG 也是未来家庭互联市场有力的竞争技术。该技术可工作在 40GHz～60GHz 的超高频段，其传输速度可以达到 1Gbps 以上，不能穿过墙壁。目前英特尔、高通等芯片企业在支持 WIGIG 发展，目前该技术还在完善中，如需要进一步降低功耗等。

12.1.3 蓝牙——4.0 进入低功耗时代

蓝牙能在包括移动电话、PDA、无线耳机、笔记本电脑、相关外设等众多设备之间进行无线信息交换。蓝牙采用分散式网络结构以及快跳频和短包技术，支持点对点及点对多点通信，工作在全球通用的 2.4GHz 频段，其数据速率为 1Mbps。

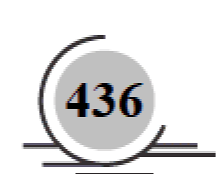
2010 年 7 月，以低功耗为特点的蓝牙 4.0 标准推出，蓝牙大中华区技术市场经理吕荣良将其看作蓝牙第二波发展高潮的标志，他表示：“蓝牙可以跨领域应用，主要有 4 个生态系统，分别是智能手机与笔记本电脑等终端市场、消费电子市场、汽车前装市场和健身运动器材市场。”

NFC 和 UWB 曾经是十分受关注的短距离无线接入技术，但其发展已经日渐势微。业内专家认为，无线频谱的规划和利用在短距离通信中日益重要。短距离通信技术目前主要采用 2.4GHz 的开放频谱，但随着物联网的发展和大量短距离通信技术的应用，频谱需求会快速增长，视频、图像等大数据量的通信正在寻求更高频段的解决方案。

12.1.4 NFC——必将逐渐远离历史舞台

NFC（Near Field Communication，近场通信）技术由非接触式射频识别（RFID）演变而来，由飞利浦半导体（现恩智浦半导体）、诺基亚和索尼共同研制开发，其基础是 RFID 及互连技术。NFC 是一种短距高频的无线电技术，在 13.56MHz 频率运行于 20 厘米距离内。其传输速度有 106Kbit/秒、212Kbit/秒或者 424Kbit/秒 3 种。目前近场通信已成为 ISO/IEC IS 18092 国际标准、ECMA-340 标准与 ETSI TS 102 190 标准。NFC 采用主动和被动两种读取模式。

NFC 近场通信技术是由非接触式射频识别（RFID）及互联互通技术整合演变而来，在单一芯片上



结合感应式读卡器、感应式卡片和点对点的功能，能在短距离内与兼容设备进行识别和数据交换。工作频率为 13.56MHz，但是使用这种手机支付方案的用户必须更换特制的手机。目前这项技术在日韩被广泛应用。手机用户凭配置了支付功能的手机就可以行遍全国：他们的手机可以用作机场登机验证、大厦的门禁钥匙、交通一卡通、信用卡、支付卡等。

NFC 和蓝牙（Bluetooth）都是短程通信技术，而且都被集成到移动电话。但 NFC 不需要复杂的设置程序。NFC 也可以简化蓝牙连接，略胜蓝牙之处在于设置程序较短，但无法达到低功耗蓝牙（Bluetooth Low Energy，BLE）的速度。在两台 NFC 设备相互连接的设备识别过程中，使用 NFC 来替代人工设置会使创建连接的速度大大加快，会少于十分之一秒。

12.2　蓝牙技术基础

蓝牙技术的数据传输速率为 1Mbps，采用时分双工传输方案实现全双工传输。本节将首先详细讲解蓝牙技术的发展历程，为读者学习本书后面的知识打下基础。

12.2.1　蓝牙技术的发展历程

蓝牙的创始者是瑞典爱立信公司，爱立信早在 1994 年就已开始进行研发。1997 年，爱立信与其他设备生产商联系，并激发了他们对该项技术的浓厚兴趣。1998 年 2 月，5 个跨国大公司，包括爱立信、诺基亚、IBM、东芝及 Intel 组成了一个特殊兴趣小组（SIG），其共同的目标是建立一个全球性的小范围无线通信技术，即现在的蓝牙。

Bluetooth 无线技术规格供全球的成员公司免费使用。许多行业的制造商都积极地在其产品中实施此技术，以减少使用零乱的电线，实现无缝连接、流传输立体声、传输数据或进行语音通信。Bluetooth 技术在 2.4GHz 波段运行，该波段是一种无须申请许可证的工业、科技、医学（ISM）无线电波段。正因如此，使用 Bluetooth 技术不需要支付任何费用，但必须向手机提供商注册使用 GSM 或 CDMA，除了设备费用外，不需要为使用 Bluetooth 技术再支付任何费用。

Bluetooth 技术得到了空前广泛的应用，集成该技术的产品从手机、汽车到医疗设备，使用该技术的用户从消费者、工业市场到企业等，不一而足。低功耗，体积小以及低成本的芯片解决方案使得 Bluetooth 技术甚至可以应用于极微小的设备中。

12.2.2　低功耗蓝牙的特点

BLE 是对传统蓝牙 BR/EDR 技术的补充。尽管 BLE 和传统蓝牙都称为蓝牙标准，且共享射频，但 BLE 是一个完全不一样的技术。BLE 不具备和传统蓝牙 BR/EDR 的兼容性，是专为小数据率、离散传输的应用而设计的。

在实际应用过程中，BLE 的低功耗并不是通过优化空中的无线射频传输实现的，而是通过改变协议的设计来实现的。为了实现极低的功耗效果，通常 BLE 协议设计为：在不必要射频时，彻底将空中射频关闭。

与传统蓝牙 BR/EDR 相比，BLE 通过如下 3 大特性实现低功耗效果。

☑ 缩短无线开启时间。

☑ 快速建立连接。

☑ 降低收发峰值功耗（具体由芯片决定）。

缩短无线开启时间的第一个技巧是只用 3 个“广告”信道，第二个技巧是通过优化协议栈来降低工作周期。一个在广告的设备可以自动和一个在搜索的设备快速建立连接，所以可以在 3 毫秒内完成连接的建立和数据的传输。

在现实应用中，低功耗设计可能会带来一些“牺牲”，例如，音频数据无法通过 BLE 来进行传输。尽管如此，BLE 仍然是一种非常出色的技术，依然会支持跳频（37 个数据信道），并且采用了一种改进的 GFSK 调制方法来提高链路的稳定性。BLE 也仍是非常安全的技术，因为在芯片级提供了 128bit AES 加密。

单模设备可以作为 Master 或者 Slave，但是不能同时充当两种角色。这意味着 BLE 只能建立简单的星状拓扑，不能实现散射网。在 BLE 的无线电规范中，定义了低功耗蓝牙的最高数据率为 305Kbps，但这只是理论数据。在实际应用中，数据的吞吐量取决于上层协议栈。而 UART 的速度、处理器的能力和主设备都会影响数据吞吐能力。

高的数据吞吐能力的 BLE 只有通过私有方案或者基于 ATT notification 才能实现。事实上，如果是高数据率或高数据量的应用，蓝牙 BR/EDR 通常显得更加省电。

12.2.3　低功耗蓝牙的架构

BLE 协议架构总体上分成 3 层，从下到上分别是控制器（Controller）、主机（Host）和应用端（Apps）。三者可以在同一芯片类中实现，也可以分不同芯片类实现，控制器（Controller）用于处理射频数据解析、接收和发送；主机（Host）用于控制不同设备之间如何进行数据交换；应用端（Apps）实现具体应用。

（1）控制器 Controller

Controller 实现射频相关的模拟和数字部分，完成最基本的数据发送和接收，包含物理层 PHY（Physical Layer）、链路层 LL（Linker Layer）、直接测试模式 DTM（Direct Test Mode）以及主机控制器接口 HCI（Host Controller Interface），其对外接口是天线，对内接口是 HCI。

☑ 物理层 PHY

GFSK 信号调制，2402MHz～2480MHz，40 个 channel，每两个 channel 间隔 2MHz（经典蓝牙协议是 1MHz），数据传输速率是 1Mbps。

☑ 链路层 LL

基于物理层 PHY 之上，实现数据通道分发、状态切换、数据包校验、加密等；链路层 LL 分两种通道：广播通道（advertising channels）和数据通道（data channels）；广播通道有 3 个，37ch（2402MHz）、38ch（2426MHz）和 39ch（2480MHz），每次广播都会向这 3 个通道同时发送（并不会在这 3 个通道之间跳频），为防止某个通道被其他设备阻塞，以至于设备无法配对或广播数据，所以定 3 个广播通道是一种权衡，少了可能会被阻塞，多了加大功耗，还有一个有意思的事情是，3 个广播通道刚好避开了 Wi-Fi 的 1ch、6ch 和 11ch，所以在 BLE 广播时，不至于被 Wi-Fi 影响（如果要干扰 BLE 广播数据，一个最简单的办法就是同时阻塞 3 个广播通道，当然不赞同这样做）；当 BLE 匹配之后，链路层 LL 由广播通道切换到数据通道，数据通道有 37 个，数据传输时会在这 37 个通道间切换，切换规则在设备间匹配时约定。

☑ 直接测试模式 DTM

为射频物理层测试接口，射频数据分析之用。

（2）主机 Host/控制器 controller 接口 HCI

HCI 作为一种接口，存在于主机 Host 和控制器 controller 当中，控制器 Host 通过 HCI 发送数据和事件给主机，主机 Host 通过 HCI 发送命令和数据给控制器 controller。HCI 逻辑上定义一系列的命令、事件；物理上有 UART、SDIO 和 USB，实际可能包含其中的任意一种或几种。

12.2.4 低功耗蓝牙分类

BLE 通常应用在传感器和智能手机或者平板电脑的通信中。到目前为止，只有很少的智能手机和平板电脑支持 BLE，例如，iPhone 4S 以后的苹果手机，Motorola Razr 和 the new iPad 及其以后的 iPad。安卓手机也逐渐支持 BLE，安卓的 BLE 标准在 2013 年 7 月 24 日刚发布。智能手机和平板电脑会带双模蓝牙的基带和协议栈，协议栈中包括 GATT 及以下的所有部分，但是没有 GATT 之上的具体协议。所以，这些具体的协议需要在应用程序中实现，实现时需要基于各个 GATT API 集。这样有利于在智能机端简单地实现具体协议，也可以在智能机端简单地开发出一套基于 GATT 的私有协议。

在现实应用中，低功耗蓝牙分为单模（Bluetooth Smart）和双模（Bluetooth Smart Ready）两种设备。BLE 和蓝牙 BR/EDR 有所区分，这样可以用 3 种方式将蓝牙技术集成到具体设备中。因为不再是所有现有的蓝牙设备可以和另一个蓝牙设备进行互联，所以准确描述产品中蓝牙的版本是非常重要的。下面将详细讲解单模蓝牙和双模蓝牙的基本知识。

（1）单模蓝牙

单模蓝牙设备被称为 Bluetooth Smart 设备，并且有专用的 Logo，如图 12-1 所示。

在现实应用中，手表、运动传感器等小型设备通常是基于低功耗单模蓝牙的。为了实现极低的功耗效果，在硬件和软件上都进行了优化，这样的设备只能支持 BLE。单模蓝牙芯片往往是一个带有单模蓝牙协议栈的产品，这个协议栈通常是芯片商免费提供的。

（2）双模蓝牙

双模蓝牙设备被称为 Bluetooth Smart Ready 设备，并且有专用的 Logo，如图 12-2 所示。



图 12-1 Bluetooth Smart 设备



图 12-2 Bluetooth Smart Ready 设备

双模设备支持蓝牙 BR/EDR 和 BLE。在双模设备中，BR/EDR 和 BLE 技术使用同一个射频前端和天线。典型的双模设备有智能手机、平板电脑、PC 和 Gateway。这些设备可以接收到通过 BLE 或者蓝牙 BR/EDR 设备发送的数据，这些设备往往都有足够的供电能力。双模设备和 BLE 设备通信的功耗低于双模设备和蓝牙 BR/EDR 设备通信的功耗。在使用双模解决方案时，需要用一个外部处理器才可以实现蓝牙协议栈。

12.2.5 集成方式

尽管有单模和双模方案的区别，但是在设备中集成蓝牙技术的方式有多种，其中最为常用的方式

有模块和芯片。

（1）模块

在现实应用中，最简单和快速的方式是使用一个嵌入式模块。此类模块包含了天线、嵌入了协议栈并提供多种不同的接口：UART、USB、SPI 和 I²C，可以通过这些接口与处理器连接。模块会提供一种简单的接口来控制蓝牙的功能。很多模块公司都会提供带 CE、FCC 和 IC 认证的产品。这样的模块可以只是蓝牙 BR/EDR 的、双模式的或者单模式的。

如果是蓝牙 BR/EDR 和双模的方案，还可以采用 HCI 模块。HCI 模块只是不带蓝牙协议栈，其他模块与上述模块是一样的。所以，这样的模块会更便宜。HCI 模块只是提供了硬件接口，在这样的方案中，蓝牙协议栈需要第三方提供。这样的第三方协议栈需要能在主设备的处理器中运行，如斯图曼提供的 BlueCode+SR。使用 HCI 模块需要将软件移植到最终的硬件中。

从理论上讲，提供单模的 HCI 模块也是可以的。然而，所有的芯片公司都已经将 GATT 集成到自己的芯片中，所以市面上不会有 HCI 单模模块出现。

（2）芯片

通过芯片来集成 BLE 是从物料角度降低成本的方式，但是这需要很多的前期工作并花费大量的时间。虽然在软件上只需要将协议栈移植到目标平台之中即可，但硬件方面则需要对 RF 的 layout 和天线的设计非常有经验。这些公司提供的 BLE 芯片有 Broadcom、CSR、EM Microelectronic、Nordic 和 TI。

12.2.6 BLE 和传统蓝牙 BR/EDR 技术的对比

BLE 和传统蓝牙 BR/EDR 技术的对比如表 12-1 所示。

表 12-1 BLE 和传统蓝牙 BR/EDR 技术的对比

	Bluetooth BR/EDR	Bluetooth Low Energy
Frequency	2400MHz～2483.5MHz	2400MHz～2483.5MHz
Deep Sleep	~80μA	<5μA
Idle	~8mA	~1mA
Peak Current	13mA～40mA	10mA～30mA
Range	500m(Class 1) / 50m (Class 2)	100m
Min. Output Power	0dBm (Class 1) / −6dBm (Class 2)	−20dBm
Max. Output Power	+20dBm (Class 1) / +4dBm (Class 2)	+10dBm
Receiver Sensitivity	⩾ −70dBm	⩾ −70dBm
Encryption	64bit/128bit	AES-128bit
Connection Time	100ms	3ms
Frequency Hopping	Yes	Yes
Advertising Channel	32	3
Data Channel	79	37
Voice capable	Yes	No

12.3 蓝牙规范详解

蓝牙规范即 Bluetooth Profile，Bluetooth SIG 定义了许多 Profile，其目的是要确保 Bluetooth 设备



间的互通性（interoperability），但是 Bluetooth 产品无须实现所有的 Bluetooth 规范。本节将详细讲解蓝牙规范的基本知识。

12.3.1　Bluetooth 系统中的常用规范

在 Bluetooth 系统中，定义了如下常用的规范。

（1）蓝牙立体声音讯传输协议 A2DP

蓝牙立体声音讯传输协议（Advance Audio Distribution Profile）的功能是播放立体声。

（2）基本图像规范

基本图像规范（Basic Imaging Profile）的功能是在装置之间传送图像，可以将其再细分为如下类别。

☑ Image Push
☑ Image Pull
☑ Advanced Image Printing
☑ Automatic Archive
☑ Remote Camera
☑ Remote Display

（3）基本打印规范

基本打印规范（Basic Printing Profile）可以将文件、电子邮件传送至打印机打印，主要包含如下分类。

☑ 无线电话规范（Cordless Telephony Profile），设置了蓝牙无线电话之间沟通的规范。
☑ 内通信规范（Intercom Profile）：是另类的 TCS（Telephone Control protocol Specification）基底规范，两个 Bluetooth 通信设备间沟通的规范。
☑ 拨号网络规范：Baseband、LMP、L2CAP、SDP 和 RFCOMM 协定所需要的传输需求。
☑ 传真规范（Fax Profile）：能传输传真的资料。
☑ 人机界面规范（Human Interface Device Profile）：可以支援鼠标和键盘功能。
☑ 头戴式通话器规范（Headset Profile）：能够将声音传送到蓝牙耳机设备。
☑ 序列埠规范（Serial Port Profile）：用来取代有线的 RS-232 Cable 。
☑ SIM 卡存取规范（SIM Access Profile）：用于存取手机内的 SIM 卡。
☑ 同步规范（Synchronization Profile）：建立在 Serial Port Profile、Generic Access Profile 与 Generic Object Exchange Profile 之上。
☑ 档案传输规范（File Transfer Profile）：Bluetooth 可以利用 OBEX 通信协定来传送档案。
☑ 泛用存取规范（Generic Access Profile）：用来建立连线。
☑ 泛用物件交换规范（Generic Object Exchange Profile）：使用 OBEX 进行物件交换。
☑ 物件交换规范（Object Push Profile）：Bluetooth 利用 OBEX 通信协定在两个设备间交换资料。
☑ 个人局域网路规范（Personal Area Networking Profile）：可以支持蓝牙网络第三层协定。
☑ 电话簿存取规范（Phone Book Access Profile）：可以在装置之间互换电话簿。
☑ 影像分享规范（Video Distribution Profile）：可以使用 H.263 编码算法来分享影像信息。

12.3.2　蓝牙协议体系结构

整个蓝牙协议体系结构可分为底层硬件模块、中间协议层和高端应用层 3 大部分。链路管理层

（LMP）、基带层（BBP）和蓝牙无线电信道构成蓝牙的底层模块。BBP 层负责跳频和蓝牙数据及信息帧的传输。LMP 层负责连接的建立和拆除以及链路的安全和控制，为上层软件模块提供了不同的访问入口，但是两个模块接口之间的消息和数据传递必须通过蓝牙主机控制器接口的解释才能进行。也就是说，中间协议层包括逻辑链路控制与适配协议（L2CAP）、服务发现协议（SDP）、串口仿真协议（RFCOMM）和电话控制协议规范（TCS）。L2CAP 完成数据拆装、服务质量控制、协议复用和组提取等功能，是其他上层协议实现的基础，因此也是蓝牙协议栈的核心部分。SDP 为上层应用程序提供一种机制来发现网络中可用的服务及其特性。在蓝牙协议栈的最上部是高端应用层，对应于各种应用模型的剖面，是剖面的一部分，目前定义了 13 种剖面。

（1）蓝牙低层模块

蓝牙的低层模块是蓝牙技术的核心，是任何蓝牙设备都必须包括的部分。蓝牙工作在 2.4GHz 的 ISM 频段。采用了蓝牙结束的设备将能够提供高达 720Kbit/s 的数据交换速率。

蓝牙支持电路交换和分组交换两种技术，分别定义了两种链路类型，即面向连接的同步链路（SCO）和面向无连接的异步链路（ACL）。为了在很低的功率状态下也能使蓝牙设备处于连接状态，蓝牙规定了 3 种节能状态，即停等（Park）状态、保持（Hold）状态和呼吸（Sniff）状态。这 3 种工作模式按照节能效率以升序排列依次是：Sniff 模式、Hold 模式、Park 模式。

蓝牙采用 3 种纠错方案，分别是 1/3 前向纠错（FEC）、2/3 前向纠错和自动重发（ARQ）。前向纠错的目的是减少重发的可能性，但同时也增加了额外开销。然而在一个合理的无错误率环境中，多余的投标会减少输出，故分组定义的本身也保持灵活的方式，因此，在软件中可定义是否采用 FEC。一般而言，在信道的噪声干扰比较大时，蓝牙系统会使用前向纠错方案，以保证通信质量：对于 SCO 链路，使用 1/3 前向纠错；对于 ACL 链路，使用 2/3 前向纠错。在无编号的自动请求重发方案中，一个时隙传送的数据必须在下一个时隙得到收到的确认消息。只有数据在接收端通过了报头错误检测和循环冗余校验（CRC）后认为无错时，才向发送端发回确认消息，否则返回一个错误消息。

蓝牙系统的移动性和开放性使得安全问题变得十分重要。虽然蓝牙系统所采用的调频技术就已经提供了一定的安全保障，但是蓝牙系统仍然需要链路层和应用层的安全管理。在链路层中，蓝牙系统提供了认证、加密和密钥管理等功能。每个用户都有一个个人标识码（PIN），它会被译成 128bit 的链路密钥（Link Key）来进行单双向认证。一旦认证完毕，链路就会以不同长度的密码（Encryphon Key）加密（此密码以 shit 为单位增减，最大的长度为 128bit），链路层安全机制提供了大量的认证方案和一个灵活的加密方案（即允许协商密码的长度）。当来自不同国家的设备互相通信时，这种机制是极其重要的，因为某些国家会指定最大密码长度。蓝牙系统会选取微微网中各个设备的最小的最大允许密码长度。例如，美国允许 128bit 的密码长度，而西班牙仅允许 48bit，这样当两国的设备互通时，将选择 48bit 来加密。蓝牙系统也支持高层协议栈的不同应用体内的特殊的安全机制。例如，两台计算机在进行商业卡信息交流时，一台计算机就只能访问另一台计算机的该项业务，而无权访问其他业务。蓝牙安全机制依赖 PIN 在设备间建立信任关系，一旦这种关系建立起来了，这些 PIN 就可以存储在设备中以便将来更快捷地连接。

（2）软件模块

L2CAP 是数据链路层的一部分，位于基带协议之上。L2CAP 向上层提供面向连接的和无连接的数据服务，其功能包括协议的复用能力、分组的分割和重新组装（Segmentation And Reaassembly）以及提取（Group Abstraction）。L2CAP 允许高层协议和应用发送和接收高达 64K Byte 的数据分组。

SDP 为应用提供了一个发现可用协议和决定这些可用协议的特性的方法。蓝牙环境下的服务发现



与传统的网络环境下的服务发现有很大的不同，在蓝牙环境下，移动的 RF 环境变化很大，因此业务的参数也是不断变换的。SDP 将强调蓝牙环境的特性。蓝牙使用基于客户/服务器机制定义了根据蓝牙服务类型和属性发现服务的方法，还提供了服务浏览的方法。

RFCOMM 是射频通信协议，可以仿真串行电缆接口协议，符合 ETSI0710 串口仿真协议。通过 RFCOMM，蓝牙可以在无线环境下实现对高层协议，如 PPP、TCP/IP、WAP 等的支持。另外，RFCOMM 可以支持 AT 命令集，从而可以实现移动电话机和传真机及调制解调器之间的无线连接。

蓝牙对语音的支持是它与 WLAN 相区别的一个重要标志。蓝牙电话控制规范是一个基于 ITU-T 建议 Q.931 的采用面向比特的协议，定义了用于蓝牙设备间建立语音和数据呼叫的呼叫控制信令以及用于处理蓝牙 TCS 设备的移动性管理过程。

12.3.3　低功耗（BLE）蓝牙协议

BLE 不再支持传统蓝牙 BR/EDR 的协议，例如，传统蓝牙中的 SPP 协议在 BLE 中就不复存在。在 BLE 应用中，所有的协议或者服务都是基于 GATT（Generic Attribute Profile）的。尽管有些传统蓝牙中的协议，如 HID 被移植到了 BLE 中，但是在 BLE 的应用中必须区分协议和服务。其中，服务描述了自身的特点和形式，并且清楚地显示了如何应用这些特点以及需要什么安全机制。而应用协议定义了其使用的服务，说明是传感器端还是接收端，定义 GATT 的角色（Server/Client）和 GAP 的角色（Peripheral/Central）。

和蓝牙 BR/EDR 协议相比，因为所有的功能都是集成在 GATT 终端，这些基于其上的应用协议只是对 GATT 提供的功能的使用，所以基于 GATT 的应用协议非常简单。

12.3.4　现有的基于 GATT 的协议/服务

截止到 2013 年 7 月，现有的基于 GATT 的协议/服务如表 12-2 所示。

表 12-2　基于 GATT 的协议/服务

GATT-Based Specifications（Qualifiable）		Adopted Version
ANP	Alert Notification Profile	1.0
ANS	Alert Notification Service	1.0
BAS	Battery Service	1.0
BLP	Blood Pressure Profile	1.0
BLS	Blood Pressure Service	1.0
CPP	Cycling Power Profile	1.0
CPS	Cycling Power Service	1.0
CSCP	Cycling Speed and Cadence Profile	1.0
CSCS	Cycling Speed and Cadence Service	1.0
CTS	Current Time Service	1.0
DIS	Device Information Service	1.1
FMP	Find Me Profile	1.0
GLP	Glucose Profile	1.0
HIDS	HID Service	1.0

续表

GATT-Based Specifications（Qualifiable）		Adopted Version
HOGP	HID over GATT Profile	1.0
HTP	Health Thermometer Profile	1.0
HTS	Health Thermometer Service	1.0
HRP	Heart Rate Profile	1.0
HRS	Heart Rate Service	1.0
IAS	Immediate Alert Service	1.0
LLS	Link Loss Service	1.0
LNP	Location and Navigation Profile	1.0
LNS	Location and Navigation Service	1.0
NDCS	Next DST Change Service	1.0
PASP	Phone Alert Status Profile	1.0
PASS	Phone Alert Status Service	1.0
PXP	Proximity Profile	1.0
RSCP	Running Speed and Cadence Profile	1.0
RSCS	Running Speed and Cadence Service	1.0
RTUS	Reference Time Update Service	1.0
ScPP	Scan Parameters Profile	1.0
ScPS	Scan Parameters Service	1.0
TIP	Time Profile	1.0
TPS	Tx Power Service	1.0

12.3.5 双模协议栈

图 12-3 中展示了斯图曼双模协议栈 BlueCode+SR 的具体架构，在此架构图中包含了 SPP、HDP 和 GATT 所需要的所有部分。

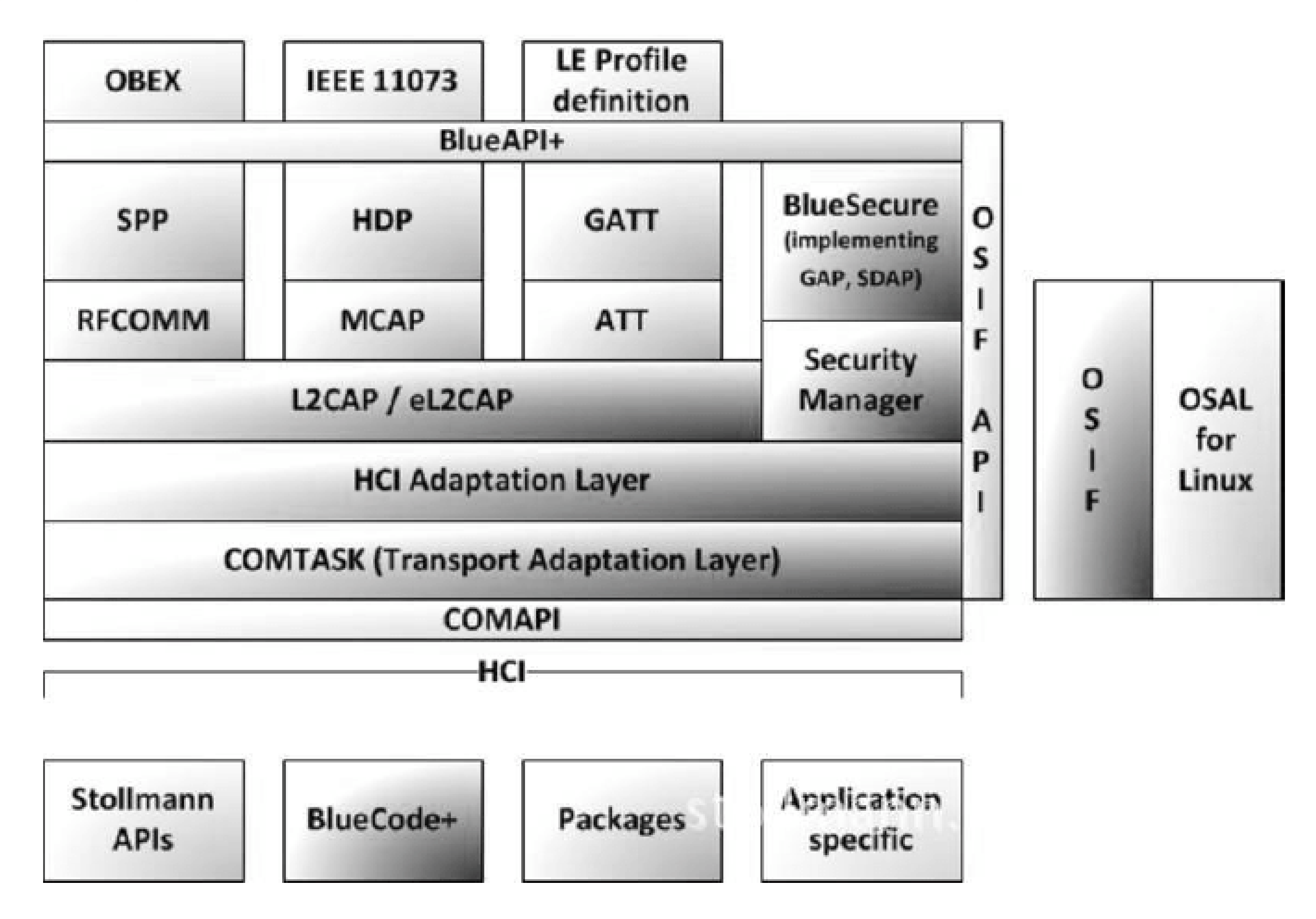


图 12-3 斯图曼双模协议栈 BlueCode+SR 的具体架构



12.3.6　单模协议栈

图 12-4 中展示了单模协议栈的一种典型协议栈设计。

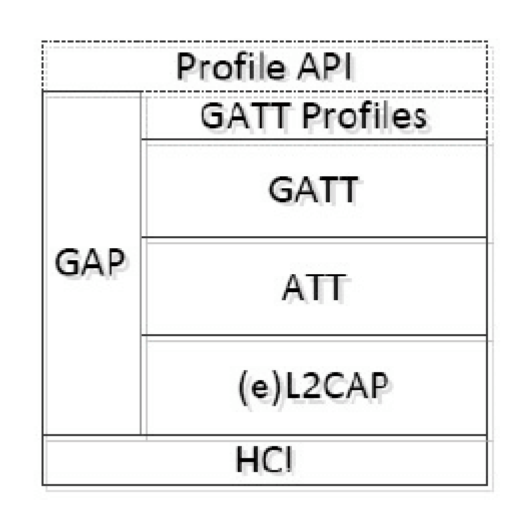


图 12-4　单模协议栈的一种设计

在单模协议栈中一般不会包含具体协议，所以需要在具体的应用程序中实现每一个具体应用对应的协议。这和传统蓝牙有很大区别，传统蓝牙会在协议栈中实现每个具体应用相关的协议，如 SPP、HDP 等。和双模协议栈相比，BLE 无须一个主处理器来实现它的协议栈，所以极低功耗的集成成为可能。大多数的单模芯片或者模块都是自带协议栈的。

因为 BLE 单模产品(芯片或者模块)中的协议栈只是实现了 GATT 层，所以通常需要将具体应用对应的协议集成到该单模产品之中。甚至芯片商都开始提供带有具体协议和 sample code 的 SDK。但是仍然没有真正能拿到手的解决方案。

12.4　低功耗蓝牙协议栈详解

在大家的印象中，提到协议栈时都会想到开放式系统互联（OSI）协议栈，OSI 协议栈定义了厂商如何生产可以与其他厂商的产品一起工作的产品。协议栈是指一组协议的集合，例如，把大象装到冰箱里需要 3 步，每步就是一个协议，3 步组成一个协议栈。本节将详细讲解低功耗蓝牙协议栈的基本知识，为读者学习本书后面的知识打下基础。

12.4.1　低功耗蓝牙协议栈基础

蓝牙协议栈就是 SIG（Special Intersted Group）定义的一组协议的规范，目标是允许遵循规范的蓝牙间应用能够进行相互操作，图 12-5 展示了完整蓝牙协议栈和部分 Profile。

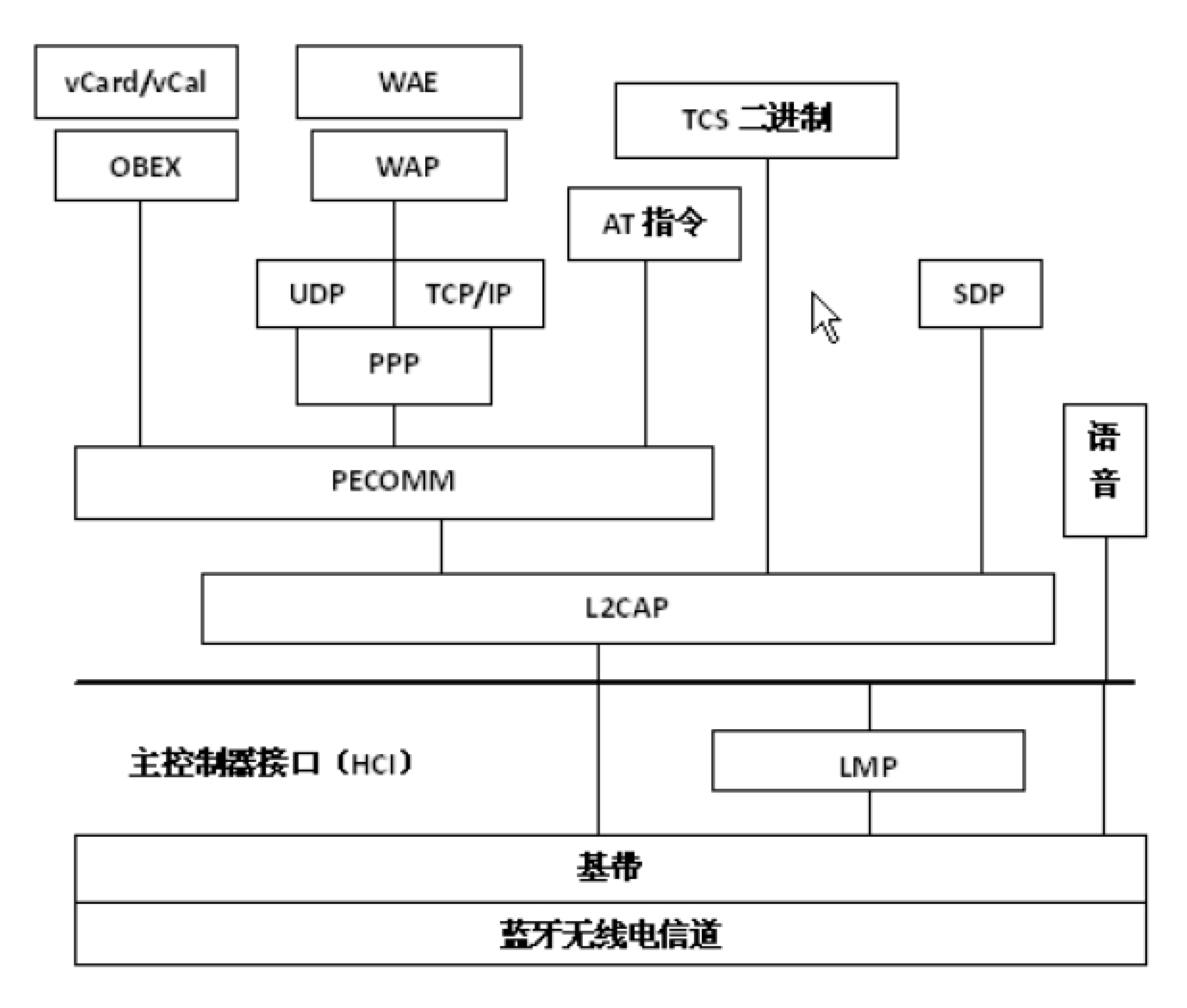


图 12-5　完整蓝牙协议栈和部分 Profile

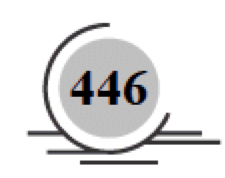
在蓝牙系统中，Profile 是配置文件，配置文件定义了可能的应用，蓝牙配置文件表达了一般行为，蓝牙设备可以通过这些行为与其他设备进行通信。在蓝牙系统中定义了广泛的配置文件，描述了许多不同类型的使用案例。按照蓝牙规格中提供的指导，开发商可以创建应用程序以与其他符合蓝牙规格的设备协同工作。到目前为止，在蓝牙系统中一共有 20 多个 Profile，在 www.bluetooth.com 中有各个 Profile 的详细说明文档。在这些众多的协议栈中，已经实现了的 Bluetooth 协议栈具体构成如下所示。

- ☑ Widcomm：第一个 Windows 上的协议栈，由 Widcomm 公司开发，也就是现在的 Broadcom。
- ☑ Microsoft Windows stack：Windows XP SP2 中包括了这个内建的协议栈，开发者也可以调用其 API 开发第三方软件。
- ☑ Toshiba stack：这也是基于 Windows 的，不支持第三方开发，但它把协议栈授权给一些 laptop 商，例如 Sony。支持的 Profile 有 SPP、DUN、FAX、LAP、OPP、FTP、HID、HCRP、PAN、BIP、HSP、 HFP、A2DP、AVRCP 和 GAVDP。
- ☑ BlueSoleil：著名的 IVT 公司的产品，该产品可以用于桌面和嵌入式，也支持第三方开发，例如 DUN、FAX、HFP、HSP、LAP、OBEX、OPP、PAN SPP、AV、BIP、FTP、GAP、HID、SDAP 和 SYNC。
- ☑ Blues：是 Linux 官方协议栈，该协议栈的上层用 Socket 封装，便于开发者使用，通过 DBUS 与其他应用程序通信。
- ☑ Affix：是 Nokia 公司的协议栈，在 Symbian 系统上运行。
- ☑ BlueDragon：是东软公司产品，在 2002 年 6 月通过了蓝牙的认证，支持的 Profile 有 SDP、Serial-DevB、AVCTP、AVRCP-Controller、AVRCP-Target、Headset-AG、Headset-HS、OPP-Client、OPP-Server、CT-GW、CT-Term、Intercom、FT-Server、FT-Client、GAP、SDAP、Serial-DevA、AVDTP、GAVDP、A2DP-Source 和 A2DP-Sink。
- ☑ BlueMagic：这是美国 Open Interface 公司针对便携式嵌入设备提供的协议栈，iPhone（Apple），nav-u（Sony）等很多电子产品都用该商业的协议栈，BlueMagic 3.0 是第一个通过 Bluetooth 协议栈 1.1 认证的协议栈。
- ☑ BCHS-Bluecore Host Software：这是蓝牙芯片 CSR 的协议栈，同时也提供了一些上层应用的 Profile 的库，当然也是为嵌入式产品提供的服务，支持的 Profile 有 A2DP、AVRCP、PBAP、BIP、BPP、CTP、DUN、FAX、FM API、FTP GAP、GAVDP、GOEP、HCRP、Headset、HF1.5、HID、ICP、JSR82、LAP Message Access Profile、OPP、PAN、SAP、SDAP、SPP、SYNC 和 SYNC ML。
- ☑ Windows CE：微软为 Windows CE 开发的协议栈，但是 Windows CE 本身也支持其他的协议栈。
- ☑ BlueLet：是 IVT 公司为嵌入式产品提供的轻量级协议栈。

12.4.2 蓝牙协议体系中的协议

在蓝牙协议体系的协议中，按 SIG 的关注程度分为如下 4 层。

- ☑ 核心协议：BaseBand、LMP、L2CAP 和 SDP。
- ☑ 电缆替代协议：RFCOMM。
- ☑ 电话传送控制协议：TCS-Binary、AT 命令集。
- ☑ 选用协议：PPP、UDP/TCP/IP、OBEX、WAP、vCard、vCal、IrMC 和 WAE。



除上述协议层外，规范还定义了主机控制器接口（HCI），它为基带控制器、连接管理器、硬件状态和控制寄存器提供命令接口。在图 12-5 中，HCI 位于 L2CAP 的下层，但 HCI 也可位于 L2CAP 上层。

蓝牙核心协议由 SIG 制定的蓝牙专用协议组成，绝大部分蓝牙设备都需要核心协议（加上无线部分），而其他协议则根据应用的需要而定。总之，电缆替代协议、电话控制协议和被采用的协议在核心协议基础上构成了面向应用的协议。

在现实应用中，常用蓝牙核心协议类型如下所示。

（1）基带协议

基带和链路控制层确保微微网内各蓝牙设备单元之间由射频构成的物理连接。蓝牙的射频系统是一个跳频系统，其任一分组在指定时隙、指定频率上发送，使用查询和分页进程同步不同设备间的发送频率和时钟，为基带数据分组提供了两种物理连接方式，即面向连接（SCO）和无连接（ACL），而且在同一射频上可实现多路数据传送。ACL 适用于数据分组，SCO 适用于话音以及话音与数据的组合，所有的话音和数据分组都附有不同级别的前向纠错（FEC）或循环冗余校验（CRC），而且可进行加密。此外，对于不同数据类型（包括连接管理信息和控制信息）都分配一个特殊通道。

可使用各种用户模式在蓝牙设备间传送话音，面向连接的话音分组只需经过基带传输，而不到达 L2CAP。话音模式在蓝牙系统内相对简单，只需开通话音连接即可传送话音。

（2）连接管理协议（LMP）

该协议负责各蓝牙设备间连接的建立，通过连接的发起、交换、核实，进行身份认证和加密，通过协商确定基带数据分组大小，还控制无线设备的电源模式和工作周期，以及微微网内设备单元的连接状态。

（3）逻辑链路控制和适配协议（L2CAP）

该协议是基带的上层协议，可以认为它与 LMP 并行工作，其区别在于，当业务数据不经过 LMP 时，L2CAP 为上层提供服务。L2CAP 向上层提供面向连接的和无连接的数据服务，采用了多路技术、分割和重组技术、群提取技术。L2CAP 允许高层协议以 64K 字节长度收发数据分组。虽然基带协议提供了 SCO 和 ACL 两种连接类型，但 L2CAP 只支持 ACL。

（4）服务发现协议（SDP）

发现服务在蓝牙技术框架中起着至关重要的作用，是所有用户模式的基础。使用 SDP 可以查询到设备信息和服务类型，从而在蓝牙设备间建立相应的连接。

（5）电缆替代协议（RFCOMM）

RFCOMM 是基于 ETSI-07.10 规范的串行线仿真协议，在蓝牙基带协议上仿真 RS-232 控制和数据信号，为使用串行线传送机制的上层协议（如 OBEX）提供服务。

（6）电话控制协议

- ☑ 二元电话控制协议（TCS-Binary 或 TCSBIN）：是面向比特的协议，定义了蓝牙设备间建立语音和数据呼叫的控制信令，定义了处理蓝牙 TCS 设备群的移动管理进程。基于 ITU TQ.931 建议的 TCSBinary 被指定为蓝牙的二元电话控制协议规范。
- ☑ AT 命令集电话控制协议：SIG 定义了控制多用户模式下移动电话和调制解调器的 AT 命令集，该 AT 命令集基于 ITU TV.250 建议和 GSM07.07，还可以用于传真业务。

（7）选用协议

- ☑ 点对点协议（PPP）：在蓝牙技术中，PPP 位于 RFCOMM 上层，完成点对点的连接。
- ☑ TCP/UDP/IP：该协议是由互联网工程任务组制定，广泛应用于互联网通信的协议。在蓝牙设

备中，使用这些协议是为了与互联网相连接的设备进行通信。

☑ 对象交换协议（OBEX）：IrOBEX（简写为 OBEX）是由红外数据协会（IrDA）制定的会话层协议，采用简单的和自发的方式交换目标。OBEX 是一种类似于 HTTP 的协议，假设传输层是可靠的，采用客户机/服务器模式，独立于传输机制和传输应用程序接口（API）。

例如，电子名片交换格式（vCard）、电子日历及日程交换格式（vCal）都是开放性规范，都没有定义传输机制，而只是定义了数据传输格式。SIG 采用 vCard/vCal 规范，是为了进一步促进个人信息交换。

☑ 无线应用协议（WAP）：该协议是由无线应用协议论坛制定的，融合了各种广域无线网络技术，其目的是将互联网内容和电话传送的业务传送到数字蜂窝电话和其他无线终端上。

12.5　TI 公司的低功耗蓝牙

BLE 低功耗蓝牙协议有很多版本，不同的厂商提供的低功耗蓝牙协议会有所区别。本节将详细讲解 TI（德州仪器）公司提供的 BLE 低功耗蓝牙协议的基本知识，为读者学习本书后面的知识打下基础。

12.5.1　获取 TI 公司的低功耗蓝牙协议栈

TI 公司提供了多个版本的 BLE 低功耗蓝牙协议栈，读者可以登录其官方网站下载，如图 12-6 所示。

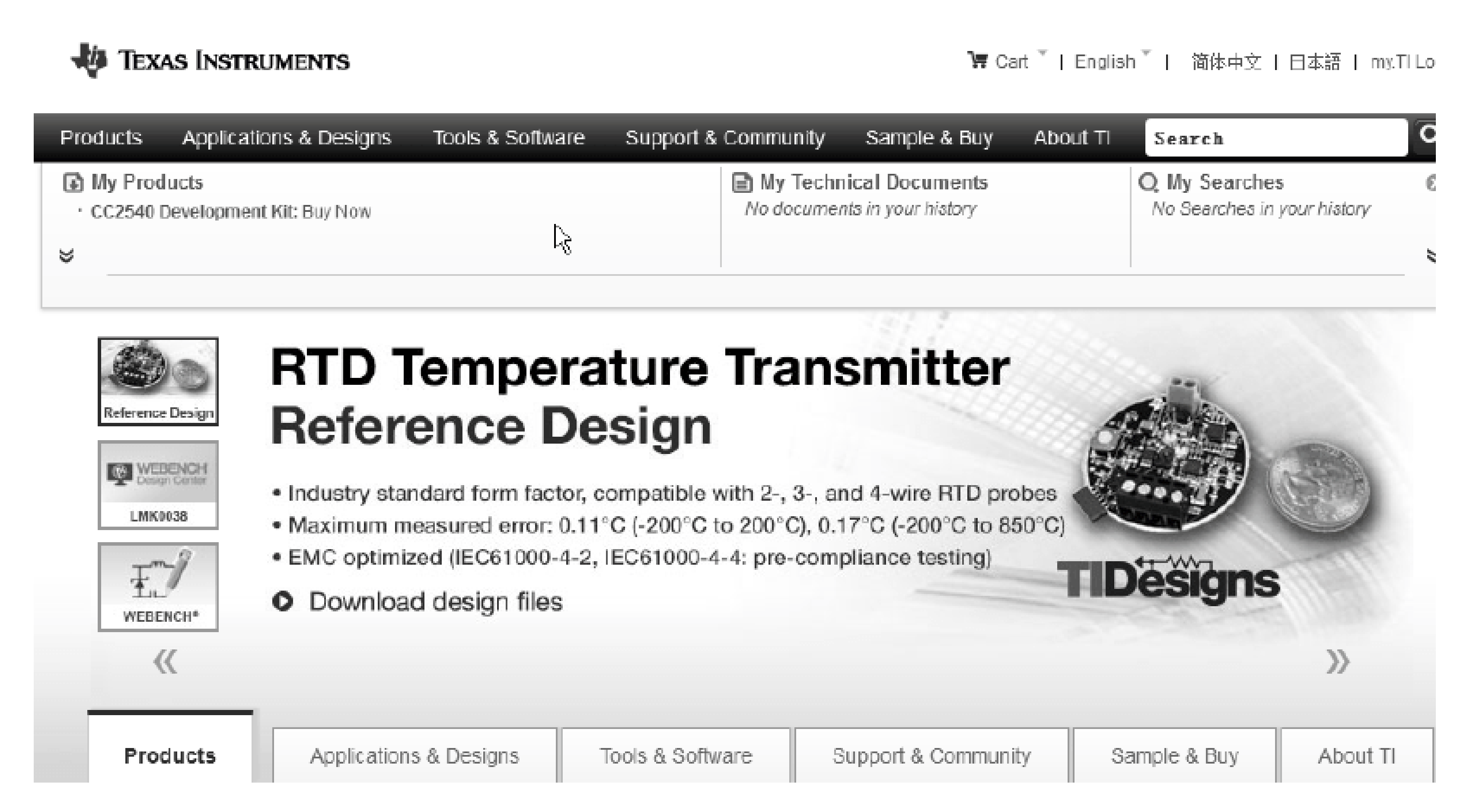


图 12-6　TI 公司的官方网站

笔者下载的版本是 BLE-CC254x-1.3.exe，双击此文件后可以进行安装工作，具体安装过程如下所示。

（1）首先弹出解压缩界面，在此单击 Next 按钮，如图 12-7 所示。

（2）弹出同意安装协议界面，在此选中 I accept…单选按钮，单击 Next 按钮，如图 12-8 所示。

（3）弹出选择安装路径界面，通过 Browse…按钮可以选择安装路径，如图 12-9 所示。

（4）弹出准备安装界面，单击 Install 按钮开始安装，如图 12-10 所示。

（5）弹出安装进度界面，此过程需要耐心等待，如图 12-11 所示。

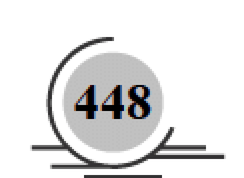




图 12-7　解压缩界面



图 12-8　同意安装协议界面

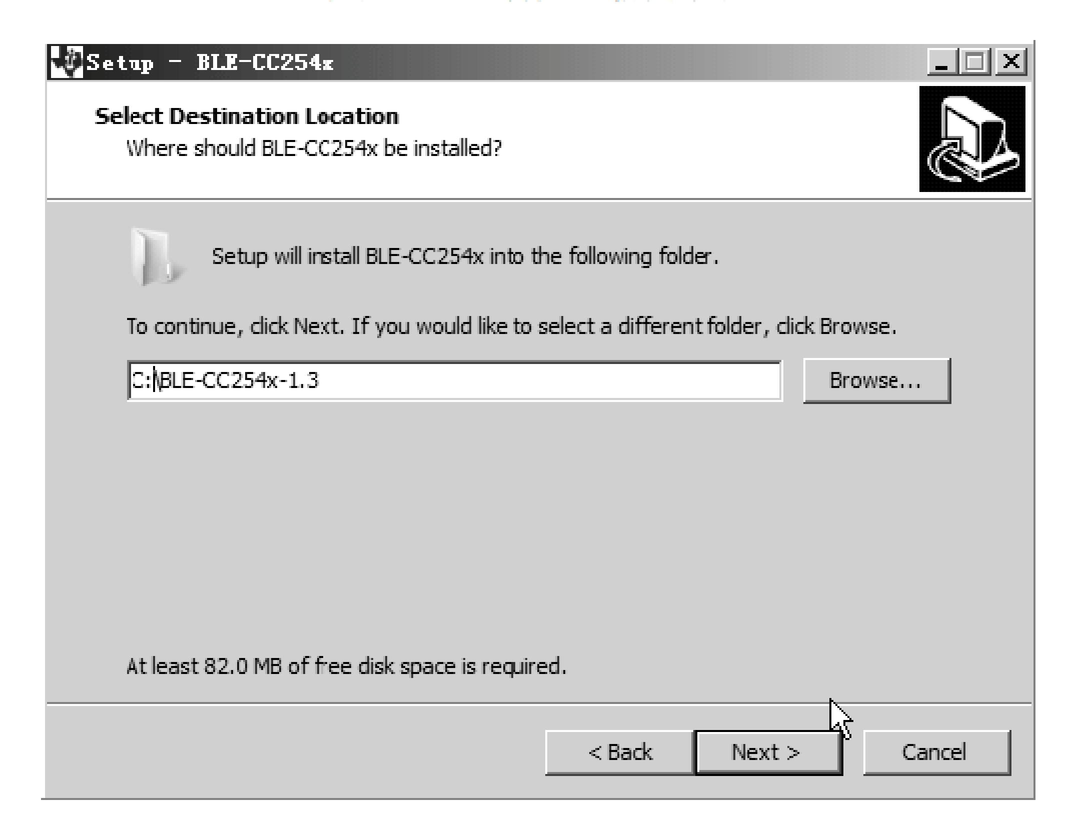


图 12-9　选择安装路径界面

Setup – BLE-CC254x
Ready to Install
Setup is now ready to begin installing BLE-CC254x on your computer.
Click Install to continue with the installation, or click Back if you want to review or change any settings.
Destination location:
C:\BLE-CC254x-1.3
< Back
Install
Cancel

图 12-10　准备安装界面

（6）最后弹出安装完成界面，整个安装过程结束，如图 12-12 所示。

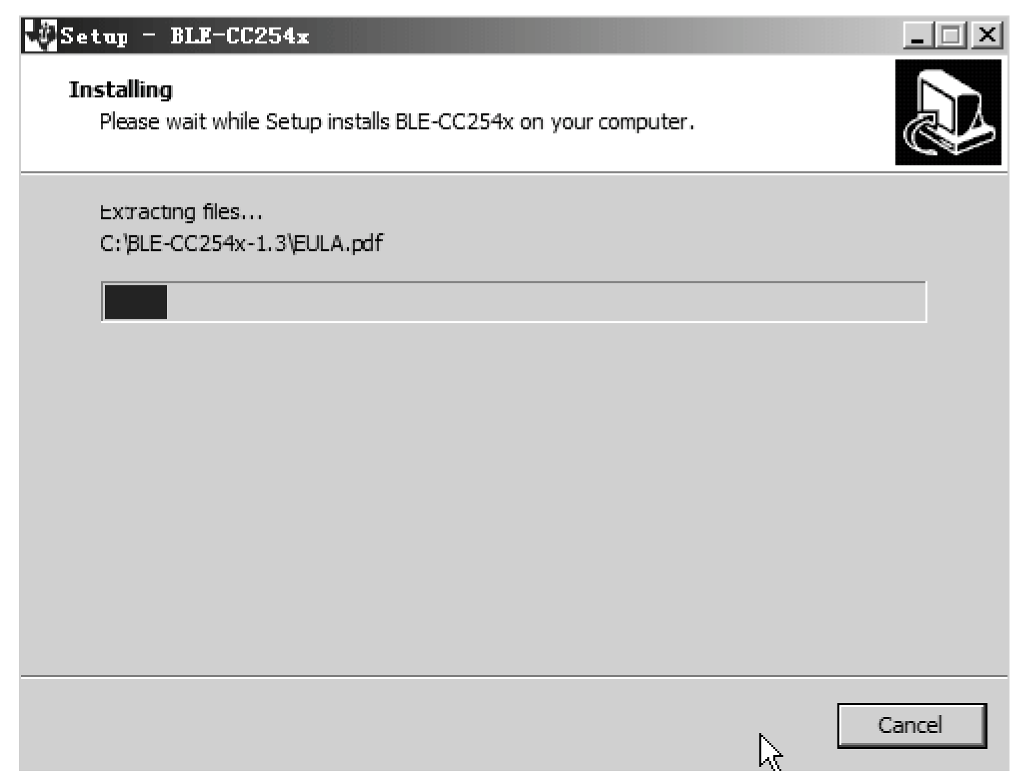


图 12-11　安装进度界面

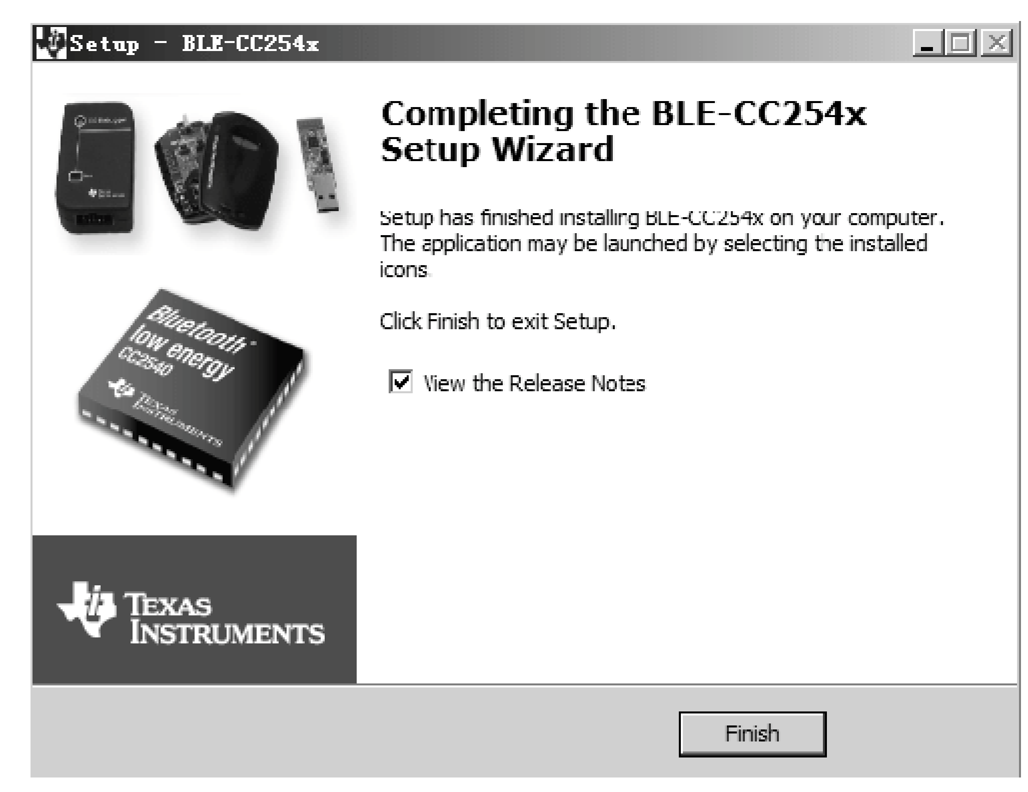


图 12-12　安装完成界面

安装完成后，需要使用 IAR 集成开发环境打开工程文件。例如，TI 公司在 Projects/ble/SimpleBLEPeripheral/目录下提供了实例工程，通过使用 IAR 工具打开.eww 文件的方式可以浏览整个工程。

注意：读者可以自行下载并安装 IAR 集成开发环境。

12.5.2 分析 TI 公司的低功耗蓝牙协议栈

注意： 下面的内容参考自 TI 公司的官方资料《CC2540Bluetooth Low Energy Software Developer's Guide (Rev. B)》，部分图片也是直接引用自上述参考文档。

1. BLE 蓝牙协议栈结构

BLE 蓝牙协议栈分为两个部分，分别是控制器和主机。对于 4.0 以前的蓝牙，这两部分是分开的。所有 Profile（暂称为“剧本”，用来定义设备或组件的角色）和应用都建构在 GAP 或 GATT 之上。BLE 蓝牙协议栈的结构如图 12-13 所示。

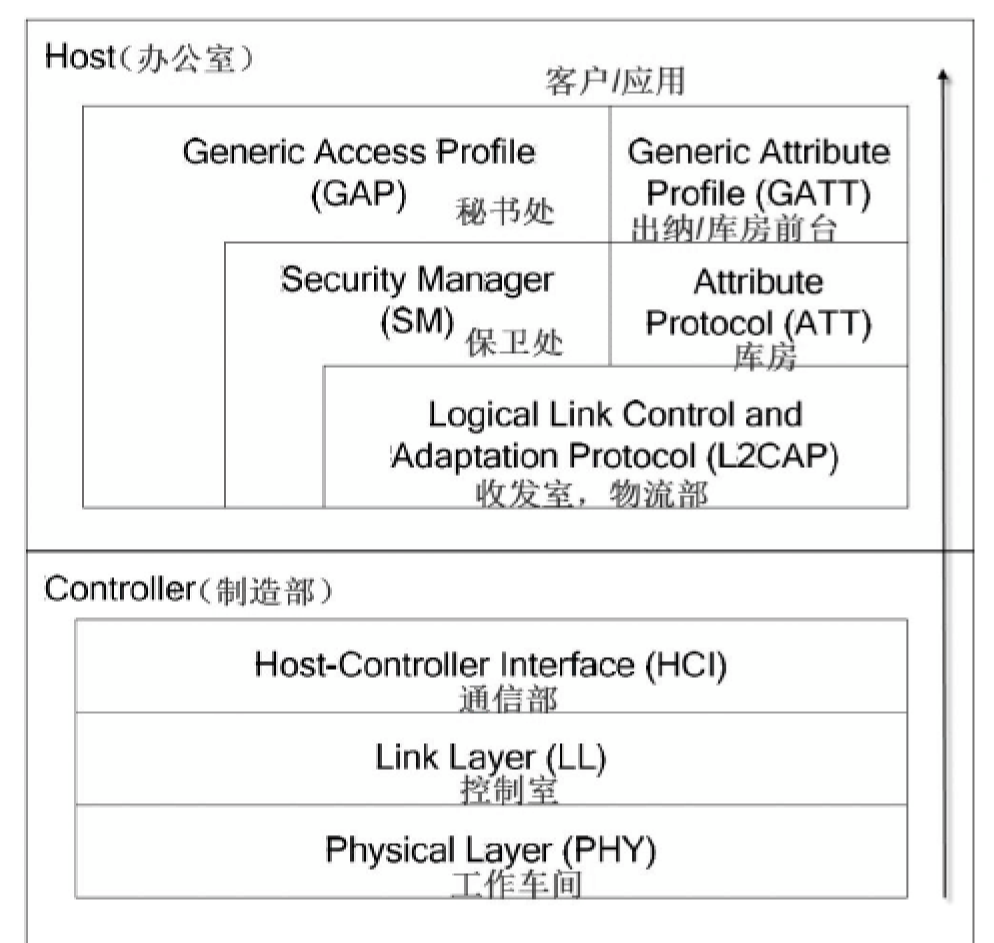


图 12-13 BLE 蓝牙协议栈的结构

在 BLE 蓝牙协议栈的结构中，从上到下的具体说明如下所示。

- ☑ PHY 层：工作车间，1Mbps 自适应跳频 GFSK（高斯频移键控），运行在免许可证使用的 2.4GHz 频段中。
- ☑ LL 层：为 RF 控制器，控制室，控制设备处于准备（standby）、广播、监听/扫描（scan）、初始化、连接这 5 种状态中的一种。5 种状态切换描述为：未连接时，设备广播信息（向周围邻居讲“我来了”），另外一个设备一直监听或按需扫描，两个设备连接初始化（搬几把椅子到院子），设备连接上了（开聊）。发起聊天的设备为主设备，接受聊天的设备为从设备，同一次聊天只能有一个意见领袖，即主设备和从设备不能切换。
- ☑ HCI 层：为接口层，通信部，向上为主机提供软件应用程序接口（API），对外为外部硬件控制接口，可以通过串口、SPI、USB 来实现设备控制。
- ☑ L2CAP 层：物流部，负责行李打包和拆封处，提供数据封装服务。
- ☑ SM 层：保卫处，提供配对和密匙分发，实现安全连接和数据交换。
- ☑ ATT 层：库房，负责数据检索。
- ☑ GATT 层：出纳/库房前台，出纳负责处理向上与应用打交道，而库房前台负责向下把检索任务子进程交给 ATT 库房去做，其关键工作是为检索工作提供合适的 profile 结构，而 profile 由检索关键词（characteristics）组成。
- ☑ GAP 层：秘书处，对上级提供应用程序接口，对下级管理各级职能部门，尤其是指示 LL 层控制室 5 种状态切换，指导保卫处做好机要工作。

蓝牙为了实现同多个设备相连或实现多功能的目标，也实现了功能扩充，这就产生了调度问题。因为虽然软件和协议栈可扩充，但终究最底层的执行部门只有一个。为了实现多事件和多任务切换，需要把事件和任务对应的应用，以及其相关的提供支撑“办公室”和“工厂”打包起来，并命令为“OSAL 操作系统抽象层”，类似于集团公司以下的子公司。

如果实现软件和硬件的低耦合，使软件不经改动或很少改动即可应用在另外的硬件上，这样就方便硬件改造、升级、迁移后软件的移植。HAL 硬件抽象层正是用来抽象各种硬件的资源，并告知给软件。其作用类似于嵌入式系统设备驱动的定义硬件资源的.h 头文件，其角色类似于现代工厂的设备管理部。

2. BLE 低功耗蓝牙系统架构

BLE 低功耗蓝牙系统架构如图 12-14 所示。

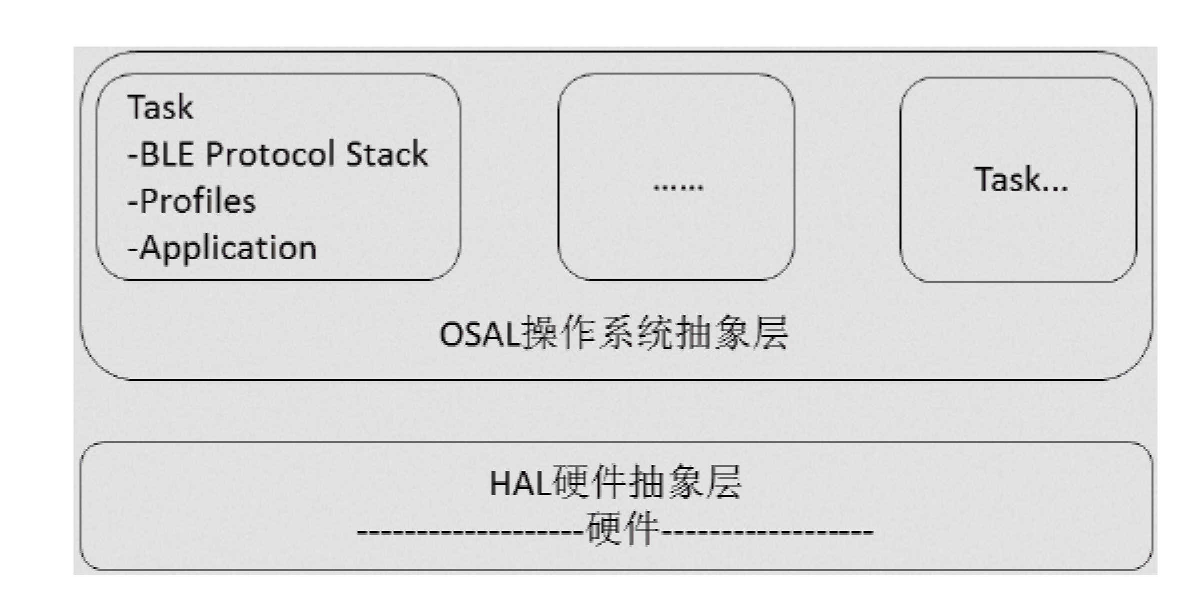


图 12-14　BLE 低功耗蓝牙系统架构图

由此可见，BLE 低功耗蓝牙软件有两个主要组成，分别是 OSAL 操作系统抽象层和 HAL 硬件抽象层，多个 Task 任务和事件在 OSAL 管理下工作，而每个任务和事件又包括 3 个组成部分，分别是 BLE 协议栈、profiles 和应用程序。

（1）OSAL 操作系统抽象层

OSAL 作为调度核心，BLE 协议栈、Profile 定义、所有的应用都围绕它来实现。OSAL 不是传统的操作系统，而是一个允许软件建立和执行事件的循环。软件功能是由任务事件来实现的，创建的任务事件需要完成如下工作。

- ☑ 创建 task identifier 任务 ID。
- ☑ 编写任务初始化（task initialization routine）进程，并需要添加到 OSAL 初始化进程中，即系统启动后不能动态添加功能。
- ☑ 编写任务处理程序。
- ☑ 提供消息服务。

BLE 协议栈的各层都是以 OSAL 任务方式实现，由于 LL 控制室的时间要求最为迫切，所以其任务优先级最高。为了实现任务管理，OSAL 通过消息处理（messageprocess）、存储管理、计时器定时等附加服务实现。

（2）系统启动流程

为了使用 OSAL，在 main()函数的最后要启动一个名叫 osal_start_system 的进程，该进程会调用由特定应用决定的启动函数 osalInitTasks()来启动系统。osalInitTasks()逐个调用 BLE 协议栈各层的启动进程来初始化协议栈。随后设置一个任务的 8bit 任务 ID（task ID），跳入循环等待执行任务，系统启动完成。

（3）任务事件与事件处理

- ☑ 进程优先级和任务 ID。
  - ➢ 任务优先级决定于任务 ID，任务 ID 越小，优先级越高。
  - ➢ BLE 协议栈各层的任务优先级比应用程序的高。
  - ➢ 初始化协议栈后，越早调入的任务，任务 ID 越高，优先级越低，即系统倾向于处理新到的任务。
- ☑ 事件变量和旗语。

每个事件任务由对应的 16bit 事件变量来标示，事件状态由旗语（taskflag）来标示。如果事件处理程序已经完成，但其旗语并没有移除，OSAL 会认为任务还没有完成而继续在该程序中不返回。例如，在

SimpleBLEPeripheral 实例工程中，当事件 START_DEVICE_EVT 发生，其处理函数 SimpleBLEPeripheral_ProcessEvent()就运行，结束后返回 16bit 事件变量，并清除旗语 SBP_START_ DEVICE_EVT。

☑ 事件处理表单。

每当 OSAL 事件检测到了有任务事件，其相应的处理进程将被添加到由处理进程指针构成的事件处理表单中，该表单名为 taskArr（taskarray）。taskArr 中各个事件进程的顺序和 osalInitTasks 初始化函数中任务 ID 的顺序是对应的。

☑ 事件调度的方法。

有两种事件调度的方法，最简单的方法是使用 osal_set_event()函数（函数原型在 OSAL.h 文件中），在这个函数中，用户可以像定义函数参数一样设置任务 ID 和事件旗语。第二种方法是使用 osal_start_timerEx()函数（函数原型在 OSAL_Timers.h 文件中），使用方法同 osal_set_event()函数，而第三个以毫秒为单位的参数 osal_start_timerEx 则指示该事件处理必须要在这个限定时间内，通过定时器来为事件处理计时。

（4）存储管理

存储管理类似于 Linux 嵌入式系统内存分配 C 函数 mem_alloc()，OSAL 利用 osal_mem_alloc 提供基本的存储管理，但 osal_mem_alloc()只有一个用于定义 byte 数的参数。对应的内存释放函数为 osal_mem_free()。

（5）进程间通信——通过消息机制实现

不同的子系统通过 OSAL 的消息机制通信。消息即为数据，数据种类和长度都不限定。消息收发的过程描述如下所示。

在接收信息时调用函数 osal_msg_allocate()创建消息占用内存空间（已经包含了 osal_mem_alloc()函数功能），需要为该函数指定空间大小，该函数返回内存空间地址指针，利用该指针即可把所需数据复制到该空间。

在发送数据时调用函数 osal_msg_send()，需为该函数指定发送目标任务，OSAL 通过旗语 SYS_EVENT_MSG 告知目标任务，目标任务的处理函数调用 osal_msg_receive 来接收发来的数据。建议每个 OSAL 任务都有一个消息处理函数，每当任务收到一个消息后，通过消息的种类来确定需要本任务做相应处理。消息接收并处理完成，调用函数 osal_msg_deallocate 来释放内存（已经包含了 osal_mem_free 函数功能）。

3. 硬件抽象层 HAL

当新的硬件平台做好后，只需修改 HAL，而不需修改 HAL 之上的协议栈的其他组件和应用程序。

4. BLE 低功耗蓝牙协议栈

（1）BLE 库文件

TI 蓝牙协议栈是以单独一个库文件提供的，并没有提供源代码，因此不做深入说明。对于 TI 的 BLE 实例应用来说，这个单独库文件完全够用，因为已经列出了所有的库文件。

（2）GAP 秘书处

☑ 角色（即服务/功能）

在 TI 实例中，GAP 运行在如下 4 种角色的一种。

➢ Broadcaster：广播员——我在，但只可远观，不可连接。



- Observer：观察员——看看谁在，但我只远观，不连接。
- Peripheral：外设（从机）——我在，谁要我就跟谁走，协议栈单层连接。
- Central：核心（主机）——看看谁在，并且谁愿意跟我走我就带谁走，协议栈单层或多层连接，目前最多支持 3 个同时连接。

虽然指标显示 BLE 可以同时扮演多个角色，但是在 TI 提供的 BLE 实例应用中默认只支持外设角色。每一种角色都由一个剧本（roleprofile）来定义。

☑ 连接

在主从机连接过程中，一个典型的低功耗蓝牙系统同时包含外设和核心（主机），两者的连接过程是：外设角色向外发送自己的信息（设备地址、名字等），主机收到外设广播信息后，发送扫描请求（scanrequest）给外设，外设响应主机的请求，连接建立完成。

连接参数主要有通信间隙（connectioninterval）、外设鄙视（slavelatency）、最大耐心等待时间（supervisiontimeout）等，具体说明如下所示。

- 通信间隙——蓝牙通信是间断的、跳频的，每次连接都可能选择不同的子频带。跳频的好处是避免频道拥塞，间断连接的好处是节省功耗，通信间隙就是指两次连接之间的时间间隔。这个间隔以 1.25ms 为基本单位，最小 6 单位，最大 3200 单位，间隙越小通信越及时，间隙越大功耗越低。
- 外设鄙视——外设与主机建立连接以后，空闲时主机总会定期发送问候信息到外设，外设不回复，这些主机发送的信息便可忽略。可以忽略的连接事件个数为 0～499 个，最多不超过 32 秒。有效连接间隙=连接间隙×（1+外设鄙视）。
- 最大耐心等待时间——指的是为了创建一个连接，主机允许的最大等候时间，在这个时间内，不停地尝试连接。范围是 10～3200 个通信间隙基本单位（1.25ms）。

以上 3 个参数大小设置优劣是显而易见的，连接参数的设置请阅读本小节后面的内容。

假如主机采用从机并不接受的参数来请求连接，又如主从机已经连接了，但从机会要求修改参数条约。通过“连接参数更新请求（Connection Parameter Update Request）”来解决问题，交由 L2CAP“收发室物流处”处理。

在实现加密处理时可以利用配对实现，利用密匙来加密授权连接。典型的过程是：外设向主机请求一个口令（passkey）以便进行配对，待主机发送了正确的口令之后，连接通信通过主从机互换密码来校验。由于蓝牙通信是间断通信，如果一个应用需要经常通信，而每次通信都要重新申请连接，那将是劳神费力的，为此 GAP 安全卫士（Security Profile，SM）提供了一种长期签证（long-termset of keys），叫做绑定（bonding），这样每次建立连接通关流程就简便并快捷。

（3）出纳 GATT

GATT 负责两个设备间通信的数据交互。共有两种角色：出纳员（GATT Client）和银行（GATT Server），银行提供资金，出纳从银行存取款。银行可以同时面对多个出纳员。这两种角色和主从机等角色是无关的。

GATT 把工作拆分成几部分来实现：读关键词（Characteristic Value）和描述符（Characteristic Descriptor），用来去库房查找提取数据，并读/写关键词和描述符。

GATT 银行（GATT Server）的业务部门（API）主要提供两个主要的功能：一是服务功能，注册或销毁服务（service attribute），并作为回调函数（callback function）；二是管理功能，添加或删除 GATT 银行业务。

一个角色定义的剧本可以同时定义多个角色，每个角色的服务、关键词、关键值、描述符（service、characteristic、characteristic value and descriptors）都以句柄（attributes）形式保存在角色提供的服务上。所有的服务都是一个 gattAttribute_t 类型的 array，在文件 gatt.h.中定义。

（4）调用 GAP 和 GATT 的一般过程

调用 GAP 和 GATT 的一般过程如下所示。

☑ API 调用。
☑ 协议栈响应并返回。
☑ 协议栈发送一个 OSAL 消息（数据）去调用相应的任务事件。
☑ 调用任务去接收和处理消息。
☑ 消息清除。

以设备初始化为 GAP 外设角色来举例说明，外设角色由其剧本（GAP peripheral role profile）来决定，实例程序在文件 peripheral.c 内。

☑ 调用 API 函数 GAP_DeviceInit()。
☑ GAP 检查后确认可以初始化，返回值为 SUCCESS (0x00)，并通知 BLE 工作。
☑ BLE 协议栈发送 OSAL 消息给外设角色剧本（peripheral roleprofile），消息内容包括要干什么（eventvalue）GAP_MSG_EVENT 和指标是什么（opcodevalue，参数）。
☑ 角色剧本的服务任务收到事件请求 SYS_EVENT_MSG，表示有消息来了。
☑ 角色剧本接收消息并查看，接着把消息数据转换（cast）为具体要执行的任务，并完成相应的工作（这里为 gapDeviceInitDoneEvent_t）。
☑ 角色剧本清除消息并返回。例如，GATT 客户端设备想从 GATT 服务器端读取数据，即 GATT 出纳想从 GATT 银行那边取点钱出来。
☑ 应用程序调用 GATT 子进程 API 函数 GATT_ReadCharValue()，传递的参数为连接句柄、关键词句柄和自身任务的 ID。
☑ GATT 答应了这个请求，返回值为 SUCCESS (0x00)，向下告知 BLE 有任务。
☑ BLE 协议栈在下次建立蓝牙连接时，发送取钱的指令给银行，当银行接收并可执行指令时，把钱取出来交给 BLE。
☑ BLE 接着就把取到的钱包成消息（OSAL message），通过 GATT 出纳返回给了应用程序。消息内包含 GATT_MSG_EVENT 和修改了的 ATT_READ_RSP。
☑ 应用程序接收到了从 OSAL 传来的 SYS_EVENT_MSG 事件，表示钱可能到了。
☑ 应用程序接收消息，拆包检查，并拿走需要的钱。
☑ 最后应用程序把包装袋销毁。

（5）GAP 角色剧本 Profiles

在 TI 的 BLE 实例应用中提供了 3 种 GAP 角色剧本，分别是保卫处角色和几种 GATT 出纳/库管示例程序服务角色。

☑ GAP 外设剧本

其 API 函数在 peripheral.h 中定义，包括如下信息。

➢ GAPROLE_ADVERT_ENABLED——广播使能。
➢ GAPROLE_ADVERT_DATA——包含在广播里的信息。
➢ GAPROLE_SCAN_RSP_DATA——外设用于回复主机扫描请求的信息。



- GAPROLE_ADVERT_OFF_TIME——表示外设关闭广播持续时间，该值为 0 表示无限期关闭广播直到下一次广播使能信号到来。
- GAPROLE_PARAM_UPDATE_ENABLE——使能自动更新连接参数，可以让外设连接失败时自动调整连接参数以便重新连接。
- GAPROLE_MIN_CONN_INTERVAL——设置最小连接间隙，默认值为 80 个单位（每单位 1.25ms）。
- GAPROLE_MIN_CONN_INTERVAL——设置最大连接间隙，默认值为 3200 个单位。
- GAPROLE_SLAVE_LATENCY——外设鄙视参数，默认值为 0。
- GAPROLE_TIMEOUT_MULTIPLIER——最大耐心等待时间，默认值为 1000 个单位。

函数 GAPRole_StartDevice()用来初始化 GAP 外设角色，其唯一的参数是 gapRolesCBs_t，这个参数是一个包含两个函数指针的结构体，这两个函数是 pfnStateChange()和 pfnRssiRead()，前者表示状态，后者表示 RSSI 已经被读走了。

☑ 多角色同时扮演

在此以设备同时为外设和广播员两种角色，方法是去除前文外设的定义剧本 peripheral.c 和 peripheral.h，添加新的剧本 peripheralBroadcaster.c 和 peripheralBroadcaster.h；定义处理器值（preprocessorvalue）PLUS_BROADCASTER。

☑ GAP 主机剧本

与外设剧本相似，主机剧本的 API 函数在 central.h 中定义，包括 GAPCentralRole_GetParameter 和 GAPCentralRole_SetParameter 以及其他。如 GAPROLE_PARAM_UPDATE_ENABLE 连接参数自动更新使能的功能，与外设角色的一样。

GAPCentralRole_StartDevice()函数用来初始化 GAP 主机角色，其唯一的参数是 gapCentralRolesCBs_t，这个参数是一个包含两个函数指针的结构体，这两个函数是 eventCB()和 rssiCB()，每次 GAP 时间发生，前者都会被调用，后者表示 RSSI 已经被读走。

☑ GAP 绑定管理器剧本

GAP 绑定管理器剧本用于保持长期的连接。同时支持外设配置和主机配置。当建立了配对连接后，如果绑定使能，绑定管理器就维护这个连接。主要参数如下。

- GAPBOND_PAIRING_MODE
- GAPBOND_MITM_PROTECTION
- GAPBOND_IO_CAPABILITIES
- GAPBOND_IO_CAP_DISPLAY_ONLY
- GAPBOND_BONDING_ENABLED

函数 GAPBondMgr_Register()用来初始化 GAP 主机角色，其唯一的参数是 gapBondCBs_t，这个参数是一个包含两个函数指针的结构体，这两个函数是 pairStateCB()和 passcodeCB()，前者返回状态，后者用于配对时产生 6 位数字口令（passcode）。

☑ 编写一个剧本来创建（定义）新的角色（功能、服务）

以 SimpleGATT Profile 为剧本名称，包含两个文件 simpleGATTProfile.c 和 simpleGATTProfile.h。包含如下主要 API 函数。

- SimpleProfile_AddService——用于初始化的进程，作用是添加服务句柄（serviceattributes）到句柄组（attributetable）内，寄存器读取和回写。

- SimpleProfile_SetParameter：设置剧本（profile）关键词（characteristics）。
- SimpleProfile_GetParameter：获取设置剧本关键词。
- SimpleProfile_RegisterAppCBs：注册 simpleProfile 回调函数。
- SimpleProfile_ReadAttrCB：读 simpleProfile 回调函数。
- SimpleProfile_WriteAttrCB：写 simpleProfile 回调函数。
- SimpleProfile_HandleConnStatusCB：连接 simpleProfile 状态函数。

此实例剧本共有如下 5 个关键词：

- SIMPLE PROFILE_CHAR1
- SIMPLE PROFIIE_CHAR2
- SIMPLE PROFIIE_CHAR3
- SIMPLE PROFIIE_CHAR4
- SIMPLE PROFILE_CHAR5

为节省本书的篇幅，TI 公司的低功耗蓝牙协议栈的基本知识介绍完毕。有关此协议栈的具体知识，请读者登录其官方网站查看帮助文档。

12.6 分析 Android 系统中的蓝牙模块

在 Android 系统中包含了对蓝牙网络协议栈的支持，这使得蓝牙设备能够无线连接其他蓝牙设备交换数据。Android 的应用程序框架提供了访问蓝牙功能的 APIs。这些 APIs 让应用程序能够无线连接其他蓝牙设备，实现点对点，或点对多点的无线交互功能。

Android 平台的蓝牙系统是基于 BlueZ，通过 Linux 中一套完整的蓝牙协议栈开源实现的。当前 BlueZ 被广泛应用于各种 Linux 版本中，并被芯片公司移植到各种芯片平台上使用。在 Linux 2.6 内核中已经包含了完整的 BlueZ 协议栈，在 Android 系统中已经移植并嵌入进了 BlueZ 的用户空间实现，并且随着硬件技术的发展而不断更新。

在 Android 系统的蓝牙模块中，除了使用 Kernel 支持外，还需要用户空间的 BlueZ 的支持。Android 系统中蓝牙模块的基本层次结构如图 12-15 所示。

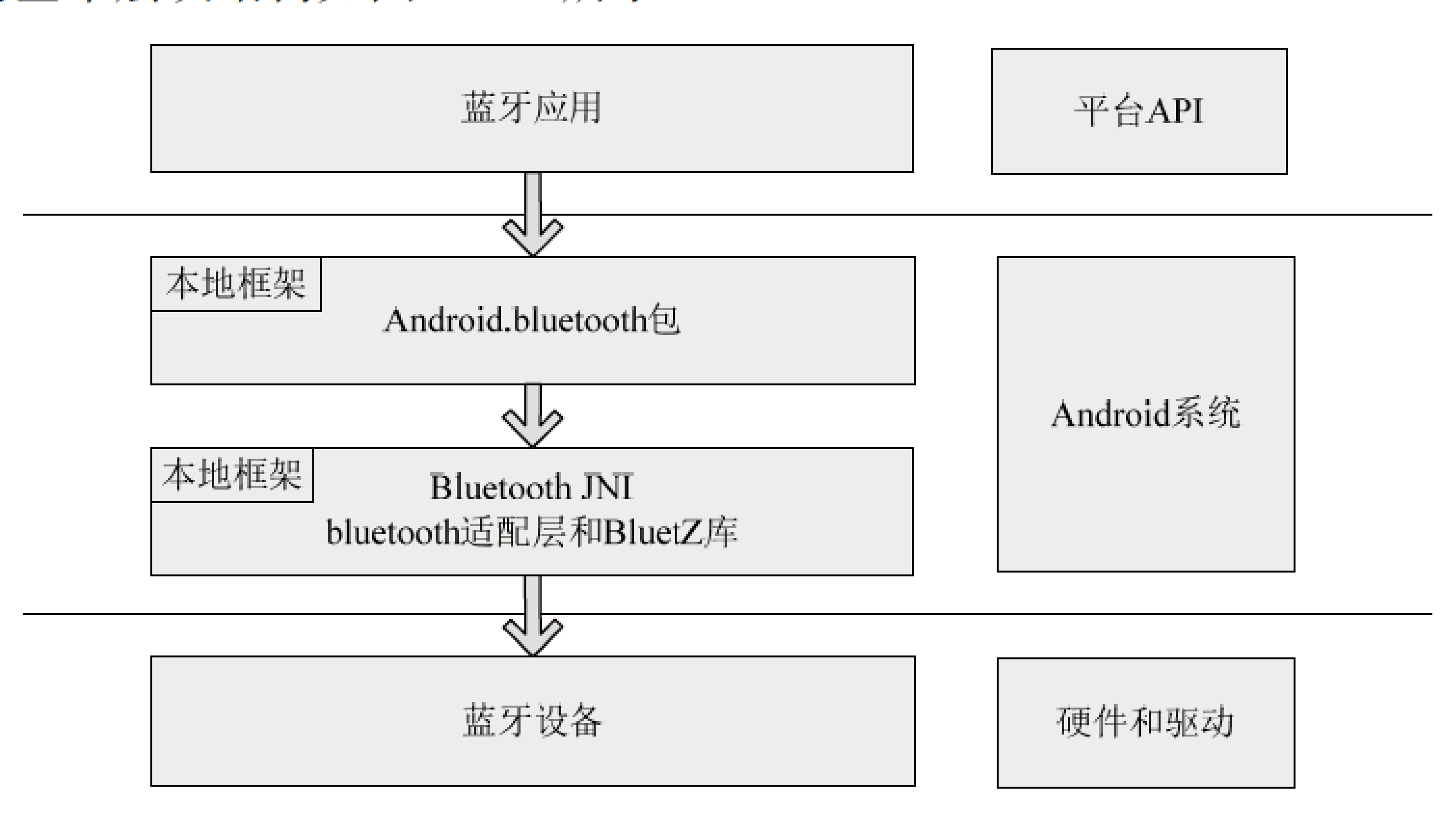


图 12-15　蓝牙系统的层次结构

Android 平台中蓝牙系统从上到下主要包括 Java 框架中的 BlueTooth 类、Android 适配库、BlueZ 库、驱动程序和协议，这几部分的系统结构如图 12-16 所示。

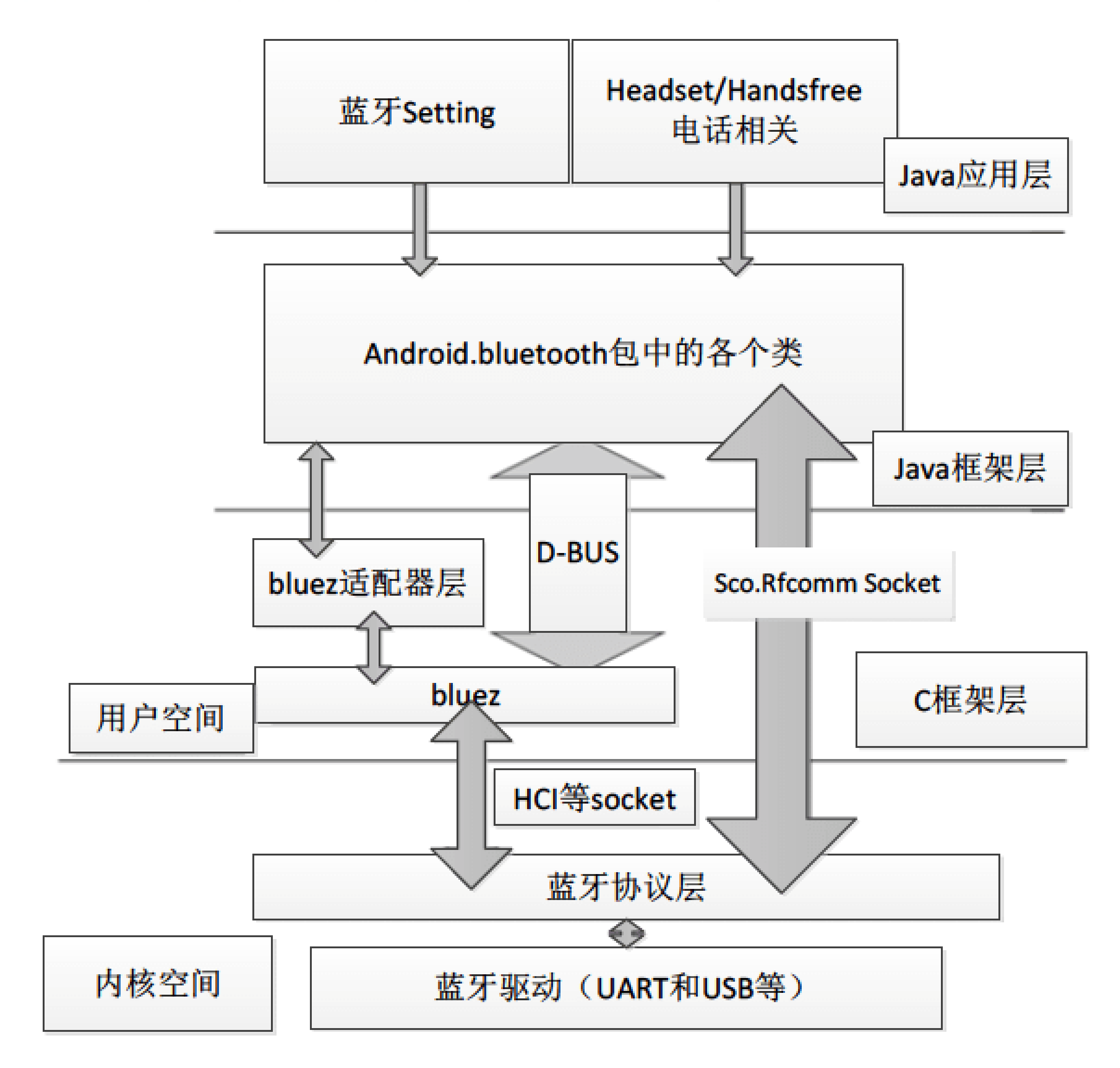


图 12-16　蓝牙系统结构

在图 12-16 中各个层次结构的具体说明如下所示。

（1）BlueZ 库

在图 12-16 所示的结构中，BlueZ 库属于 C 框架层，Android 蓝牙设备管理的库的路径是 external/bluez/。

可以分别生成 libbluetooth.so、libbluedroid.so 和 hcidump 等众多相关工具和库。BlueZ 库提供了对用户空间蓝牙的支持，其中包含了主机控制协议 HCI 以及其他众多内核实现协议的接口，并且实现了所有蓝牙应用模式 Profile。

（2）蓝牙的 JNI 部分

在图 12-16 所示的结构中，蓝牙 JNI 部分属于 C 框架层，此部分的代码路径是 frameworks/base/core/jni/。

（3）Java 框架层

Java 框架层的实现代码保存在如下路径。

☑　frameworks/base/core/java/android/bluetooth：蓝牙部分对应应用程序的 API。

☑　frameworks/base/core/java/android/Server：蓝牙的服务部分。

蓝牙的服务部分负责管理并使用底层本地服务，并封装成系统服务。而在 android.bluetooth 部分中包含了各个蓝牙平台的 API 部分，以供应用程序层使用。

（4）Bluetooth 的适配库

在图 12-16 所示的结构中，Bluetooth 的适配库属于 C 框架层，功能是在用户空间和 Java 框架之间搭建一个桥梁。Bluetooth 适配库的代码路径是 system/bluetooth/。

在此层用于生成库 libbluedroid.so 以及相关工具和库，能够实现对蓝牙设备的管理，例如蓝牙设备的电源管理。

12.7 分析蓝牙模块的源码

要想掌握蓝牙系统的开发原理，需要首先分析 Android 中的蓝牙源码并了解其核心构造，只有这样才能对蓝牙应用开发做到游刃有余。本节简要介绍开源 Android 中蓝牙模块相关的代码，为读者学习本书后面的知识打下基础。

12.7.1 初始化蓝牙芯片

初始化蓝牙芯片工作是通过 BlueZ 工具 hciattach 进行的，此工具在目录 external/bluetooth/tools 的文件中实现。

hciattach 命令主要用来初始化蓝牙设备，其命令格式如下所示。

```
hciattach [-n] [-p] [-b] [-t timeout] [-s initial_speed] <tty> <type | id> [speed] [flow|noflow] [bdaddr]
```

在上述格式中，最重要的参数就是 type 和 speed，type 决定了要初始化的设备的型号，可以使用 hciattach -1 列出所支持的设备型号。

并不是所有的参数对所有的设备都是适用的，有些设备会忽略一些参数设置，例如，查看 hciattach 的代码就可以看到，多数设备都忽略 bdaddr 参数。hciattach 命令内部的工作步骤是：首先打开制定的 tty 设备，然后做一些通用的设置，如 flow 等，然后设置波特率为 initial_speed，然后根据 type 调用各自的初始化代码，最后将波特率重新设置为 speed。所以调用 hciattach 时，要根据实际情况设置好 initial_speed 和 speed。

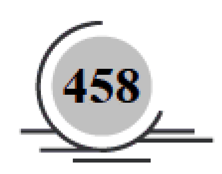
对于 type BCSP 来说，其初始化代码只做了一件事，就是完成 BCSP 协议的同步操作，并不对蓝牙芯片做任何 pskey 的设置。

12.7.2 蓝牙服务

在蓝牙服务方面一般不要自己定义，只需要使用初始化脚本文件 init.rc 中的默认内容即可。例如下面的代码。

```
service bluetoothd /system/bin/logwrapper /system/bin/bluetoothd -d -n
    socket bluetooth stream 660 bluetooth bluetooth
    socket dbus_bluetooth stream 660 bluetooth bluetooth
    # init.rc does not yet support applying capabilities, so run as root and
    # let bluetoothd drop uid to bluetooth with the right linux capabilities
    group bluetooth net_bt_admin misc
    disabled

# baudrate change 115200 to 1152000(Bluetooth)
service changebaudrate /system/bin/logwrapper /system/xbin/bccmd_115200 -t bcsp -d /dev/s3c2410_serial1
psset -r 0x1be 0x126e
    user bluetooth
    group bluetooth net_bt_admin
```



```
    disabled
    oneshot

#service hciattach /system/bin/logwrapper /system/bin/hciattach -n -s 1152000 /dev/s3c2410_serial1 bcsp 1152000
service hciattach /system/bin/logwrapper /system/bin/hciattach -n -s 115200 /dev/s3c2410_serial1 bcsp 115200
    user bluetooth
    group bluetooth net_bt_admin misc
    disabled

service hfag /system/bin/sdptool add --channel=10 HFAG
    user bluetooth
    group bluetooth net_bt_admin
    disabled
    oneshot

service hsag /system/bin/sdptool add --channel=11 HSAG
    user bluetooth
    group bluetooth net_bt_admin
    disabled
    oneshot

service opush /system/bin/sdptool add --channel=12 OPUSH
    user bluetooth
    group bluetooth net_bt_admin
    disabled
    oneshot

service pbap /system/bin/sdptool add --channel=19 PBAP
    user bluetooth
    group bluetooth net_bt_admin
    disabled
    oneshot
```

在上述代码中，每一个 service 后面都列出了一种 Android 服务。

12.7.3　管理蓝牙电源

在 Android 系统的 system/bluetooth/目录中实现了 libbluedroid。

可以调用 rfkill 接口来控制电源管理，如果已经实现了 rfkill 接口，则无须再进行配置。如果在文件 init.rc 中已经实现了 hciattach 服务，则说明在 libbluedroid 中已经实现对其调用以操作蓝牙的初始化。

12.8　Android 系统的低功耗蓝牙协议栈

从 Android 4.2 版本开始，Google 便更换了 Android 的蓝牙协议栈，从 BlueZ 换成 BlueDroid。从 Android 4.3 版本开始，提供了对蓝牙 4.0 BLE 的支持。本节将详细讲解 Android 系统中的蓝牙 4.0 BLE 的基本知识，为读者学习本书后面的知识打下基础。

12.8.1　Android 低功耗蓝牙协议栈基础

为了确保 Android 系统可以更好地支持蓝牙 4.0 BLE，Broadcom 公司特意推出了适应于 Android 平台的开源低功耗蓝牙协议栈 BlueDroid，其开发文档和 API 是开源代码，在地址 https://github.com/briandbl/framework 中保存。

在上述开源代码中，低功耗蓝牙 API 支持 Android 平台上的低功耗蓝牙通信功能。通过使用 BlueDroid 协议栈，Android 应用程序可以枚举、发现并访问低功耗蓝牙的外部设备，并且实现了低功耗蓝牙规范。

从 Android 4.2 版本开始，低功耗蓝牙模块的整体结构如图 12-17 所示。

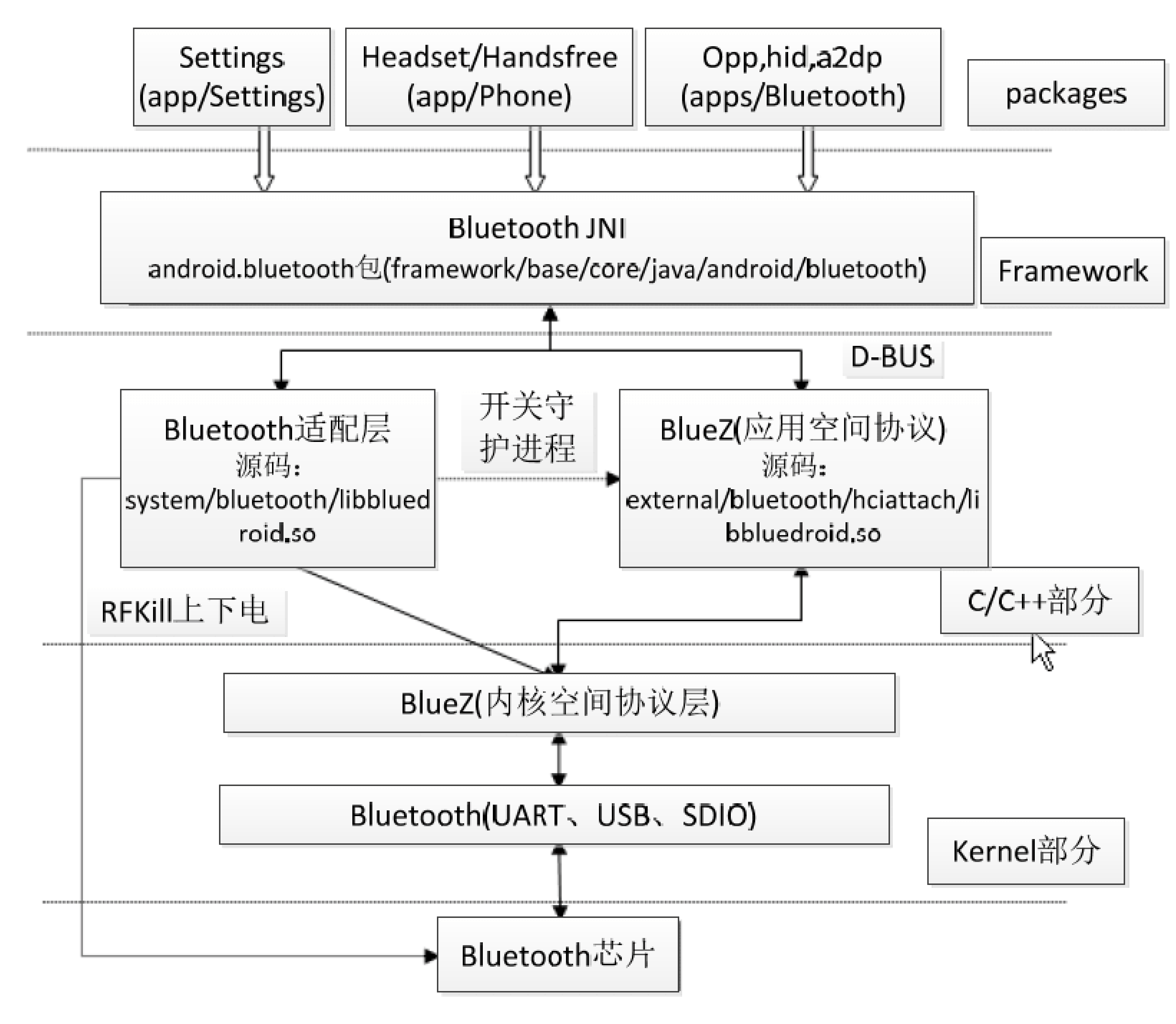


图 12-17　低功耗蓝牙模块的整体结构

注意： 虽然从 Android 4.2 版本开始，JNI 部分的代码在 packages 层中实现。但是为了便于读者从视觉上更加容易接受，所以将 JNI 部分绘制在了 Framework 层中。

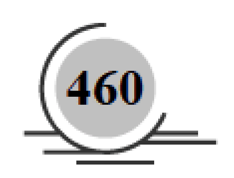
12.8.2　低功耗蓝牙 API 详解

Broadcom 公司推出的低功耗蓝牙协议栈 BlueDroid 的开发文档和 API 是开源代码，被保存在地址 https://github.com/briandbl/framework 中。

下面将详细讲解主要 API 的基本功能和具体原理。

（1）本地蓝牙适配器设备

本功能不是由 Broadcom 公司提供的，而是由 Android SDK 提供的，源码位于目录 framework/base/



core/java/android.bluetooth/BluetoothAdapter.java 中。

文件 BluetoothAdapter.java 实现了所有蓝牙交互的入口。通过使用类 BluetoothAdapter 可以实现如下功能。

☑　发现其他的蓝牙设备，查询匹配的设备集。
☑　使用一个已知蓝牙地址来初始化蓝牙设备 BluetoothDevice。
☑　创建一个能够监听其他设备通信的类 BluetoothSocket。

文件 BluetoothAdapter.java 的主要实现代码如下所示。

```
public static synchronized BluetoothAdapter getDefaultAdapter() {
    if (sAdapter == null) {
        IBinder b = ServiceManager.getService(BLUETOOTH_MANAGER_SERVICE);
        if (b != null) {
            IBluetoothManager managerService = IBluetoothManager.Stub.asInterface(b);
            sAdapter = new BluetoothAdapter(managerService);
        } else {
            Log.e(TAG, "Bluetooth binder is null");
        }
    }
    return sAdapter;
}
BluetoothAdapter(IBluetoothManager managerService) {
     if (managerService == null) {
        throw new IllegalArgumentException("bluetooth manager service is null");
    }
    try {
        mService = managerService.registerAdapter(mManagerCallback);
    } catch (RemoteException e) {Log.e(TAG, "", e);}
    mManagerService = managerService;
    mLeScanClients = new HashMap<LeScanCallback, GattCallbackWrapper>();
}
public BluetoothDevice getRemoteDevice(byte[] address) {
    if (address == null || address.length != 6) {
        throw new IllegalArgumentException("Bluetooth address must have 6 bytes");
    }
    return new BluetoothDevice(String.format("%02X:%02X:%02X:%02X:%02X:%02X",
            address[0], address[1], address[2], address[3], address[4], address[5]));
}
public int getState() {
    try {
        synchronized(mManagerCallback) {
            if (mService != null)
            {
                int state=   mService.getState();
                if (VDBG) Log.d(TAG, "" + hashCode() + ": getState(). Returning " + state);
                return state;
            }
        }
    } catch (RemoteException e) {Log.e(TAG, "", e);}
    if (DBG) Log.d(TAG, "" + hashCode() + ": getState() :   mService = null. Returning STATE_OFF");
```

```
        return STATE_OFF;
}
public String getAddress() {
        try {
                return mManagerService.getAddress();
        } catch (RemoteException e) {Log.e(TAG, "", e);}
        return null;
}
public String getName() {
        try {
                return mManagerService.getName();
        } catch (RemoteException e) {Log.e(TAG, "", e);}
        return null;
}

public int getScanMode() {
        if (getState() != STATE_ON) return SCAN_MODE_NONE;
        try {
                synchronized(mManagerCallback) {
                        if (mService != null) return mService.getScanMode();
                }
        } catch (RemoteException e) {Log.e(TAG, "", e);}
        return SCAN_MODE_NONE;
}
  */
public boolean setScanMode(int mode, int duration) {
        if (getState() != STATE_ON) return false;
        try {
                synchronized(mManagerCallback) {
                        if (mService != null) return mService.setScanMode(mode, duration);
                }
        } catch (RemoteException e) {Log.e(TAG, "", e);}
        return false;
}

/** @hide */
public boolean setScanMode(int mode) {
        if (getState() != STATE_ON) return false;
        /* getDiscoverableTimeout() to use the latest from NV than use 0 */
        return setScanMode(mode, getDiscoverableTimeout());
}

/** @hide */
public int getDiscoverableTimeout() {
        if (getState() != STATE_ON) return -1;
        try {
                synchronized(mManagerCallback) {
                        if (mService != null) return mService.getDiscoverableTimeout();
                }
        } catch (RemoteException e) {Log.e(TAG, "", e);}
        return -1;
```

```
}

/** @hide */
public void setDiscoverableTimeout(int timeout) {
    if (getState() != STATE_ON) return;
    try {
        synchronized(mManagerCallback) {
            if (mService != null) mService.setDiscoverableTimeout(timeout);
        }
    } catch (RemoteException e) {Log.e(TAG, "", e);}
}
public boolean startDiscovery() {
    if (getState() != STATE_ON) return false;
    try {
        synchronized(mManagerCallback) {
            if (mService != null) return mService.startDiscovery();
        }
    } catch (RemoteException e) {Log.e(TAG, "", e);}
    return false;
}
public boolean cancelDiscovery() {
    if (getState() != STATE_ON) return false;
    try {
        synchronized(mManagerCallback) {
            if (mService != null) return mService.cancelDiscovery();
        }
    } catch (RemoteException e) {Log.e(TAG, "", e);}
    return false;
}
public boolean isDiscovering() {
    if (getState() != STATE_ON) return false;
    try {
        synchronized(mManagerCallback) {
            if (mService != null ) return mService.isDiscovering();
        }
    } catch (RemoteException e) {Log.e(TAG, "", e);}
    return false;
}
public Set<BluetoothDevice> getBondedDevices() {
    if (getState() != STATE_ON) {
        return toDeviceSet(new BluetoothDevice[0]);
    }
    try {
        synchronized(mManagerCallback) {
            if (mService != null) return toDeviceSet(mService.getBondedDevices());
        }
        return toDeviceSet(new BluetoothDevice[0]);
    } catch (RemoteException e) {Log.e(TAG, "", e);}
    return null;
}
public int getConnectionState() {
```

```
    if (getState() != STATE_ON) return BluetoothAdapter.STATE_DISCONNECTED;
    try {
        synchronized(mManagerCallback) {
            if (mService != null) return mService.getAdapterConnectionState();
        }
    } catch (RemoteException e) {Log.e(TAG, "getConnectionState:", e);}
    return BluetoothAdapter.STATE_DISCONNECTED;
}
public int getProfileConnectionState(int profile) {
    if (getState() != STATE_ON) return BluetoothProfile.STATE_DISCONNECTED;
    try {
        synchronized(mManagerCallback) {
            if (mService != null) return mService.getProfileConnectionState(profile);
        }
    } catch (RemoteException e) {
        Log.e(TAG, "getProfileConnectionState:", e);
    }
    return BluetoothProfile.STATE_DISCONNECTED;
}
public BluetoothServerSocket listenUsingRfcommOn(int channel) throws IOException {
    BluetoothServerSocket socket = new BluetoothServerSocket(
            BluetoothSocket.TYPE_RFCOMM, true, true, channel);
    int errno = socket.mSocket.bindListen();
    if (errno != 0) {
        throw new IOException("Error: " + errno);
    }
    return socket;
}
public BluetoothServerSocket listenUsingRfcommWithServiceRecord(String name, UUID uuid)
        throws IOException {
    return createNewRfcommSocketAndRecord(name, uuid, true, true);
}
private BluetoothServerSocket createNewRfcommSocketAndRecord(String name, UUID uuid,
        boolean auth, boolean encrypt) throws IOException {
    BluetoothServerSocket socket;
    socket = new BluetoothServerSocket(BluetoothSocket.TYPE_RFCOMM, auth,
                    encrypt, new ParcelUuid(uuid));
    socket.setServiceName(name);
    int errno = socket.mSocket.bindListen();
    if (errno != 0) {
        throw new IOException("Error: " + errno);
    }
    return socket;
}
```

在使用蓝牙 BLE 之前，需要确认 Android 设备是否支持 BLE feature（required 为 false 时），另外需要确认蓝牙是否打开。 如果发现不支持 BLE，则不能使用 BLE 相关的功能。如果支持 BLE，但是蓝牙没有打开，则需要打开蓝牙。打开蓝牙的基本步骤如下所示。

☑ 获取 BluetoothAdapter

BluetoothAdapter 是 Android 系统中所有蓝牙操作都需要的，对应本地 Android 设备的蓝牙模块，



在整个系统中 BluetoothAdapter 是单例的。当获取 BluetoothAdapter 的示例之后，就能进行相关的蓝牙操作了。例如，获取 BluetoothAdapter 的演示代码如下所示。

```
final BluetoothManager bluetoothManager =
        (BluetoothManager) getSystemService(Context.BLUETOOTH_SERVICE);
mBluetoothAdapter = bluetoothManager.getAdapter();
```

这样通过 getSystemService 获取 BluetoothManager，再通过 BluetoothManager 获取 BluetoothAdapter。BluetoothManager 在 Android 4.3 以上的版本中支持（API level 18）。

☑　判断是否支持蓝牙，并打开蓝牙

获取 BluetoothAdapter 之后，还需要判断是否支持蓝牙，以及蓝牙是否打开。如果没有打开，需要让用户打开蓝牙。例如下面的演示代码。

```
private BluetoothAdapter mBluetoothAdapter;
...
if (mBluetoothAdapter == null || !mBluetoothAdapter.isEnabled()) {
    Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
    startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
}
```

在启动 BLE 蓝牙设备后，需要搜索附近的蓝牙。通过调用类 BluetoothAdapter 中的方法 startLeScan() 可以搜索 BLE 设备，在调用此方法时需要传入 BluetoothAdapter.LeScanCallback 参数，需要实现 BluetoothAdapter.LeScanCallback 接口，BLE 设备的搜索结果将通过这个 callback 返回。函数 startLeScan() 的具体实现代码如下所示。

```
public boolean startLeScan(final UUID[] serviceUuids, final LeScanCallback callback) {
    if (DBG) Log.d(TAG, "startLeScan(): " + serviceUuids);
    if (callback == null) {
        if (DBG) Log.e(TAG, "startLeScan: null callback");
        return false;
    }
    BluetoothLeScanner scanner = getBluetoothLeScanner();
    if (scanner == null) {
        if (DBG) Log.e(TAG, "startLeScan: cannot get BluetoothLeScanner");
        return false;
    }

    synchronized(mLeScanClients) {
        if (mLeScanClients.containsKey(callback)) {
            if (DBG) Log.e(TAG, "LE Scan has already started");
            return false;
        }

        try {
            IBluetoothGatt iGatt = mManagerService.getBluetoothGatt();
            if (iGatt == null) {
                return false;
            }
```

```
            ScanCallback scanCallback = new ScanCallback() {
                @Override
                public void onScanResult(int callbackType, ScanResult result) {
                    if (callbackType != ScanSettings.CALLBACK_TYPE_ALL_MATCHES) {
                        Log.e(TAG, "LE Scan has already started");
                        return;
                    }
                    ScanRecord scanRecord = result.getScanRecord();
                    if (scanRecord == null) {
                        return;
                    }
                    if (serviceUuids != null) {
                        List<ParcelUuid> uuids = new ArrayList<ParcelUuid>();
                        for (UUID uuid : serviceUuids) {
                            uuids.add(new ParcelUuid(uuid));
                        }
                        List<ParcelUuid> scanServiceUuids = scanRecord.getServiceUuids();
                        if (scanServiceUuids == null || !scanServiceUuids.containsAll(uuids)) {
                            if (DBG) Log.d(TAG, "uuids does not match");
                            return;
                        }
                    }
                    callback.onLeScan(result.getDevice(), result.getRssi(),
                            scanRecord.getBytes());
                }
            };
            ScanSettings settings = new ScanSettings.Builder()
                .setCallbackType(ScanSettings.CALLBACK_TYPE_ALL_MATCHES)
                .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY).build();

            List<ScanFilter> filters = new ArrayList<ScanFilter>();
            if (serviceUuids != null && serviceUuids.length > 0) {
                ScanFilter filter = new ScanFilter.Builder().setServiceUuid(
                        new ParcelUuid(serviceUuids[0])).build();
                filters.add(filter);
            }
            scanner.startScan(filters, settings, scanCallback);

            mLeScanClients.put(callback, scanCallback);
            return true;

        } catch (RemoteException e) {
            Log.e(TAG,"",e);
        }
    }
    return false;
}
```

其中，参数 UUID 数组指定了应用程序所支持的 GATT Services 的 UUID。在搜索时只能搜索传统蓝牙设备或者 BLE 设备，两者完全独立，不可同时被搜索。



由于搜索需要尽量减少功耗，因此在实际使用时需要注意如下两点。

☑ 当找到对应的设备后，立即停止扫描。

☑ 不要循环搜索设备，为每次搜索设置适合的时间限制。避免设备不在可用范围时持续不停地扫描，消耗电量。

（2）请求远程蓝牙设备

本功能也不是由 Broadcom 公司提供的，而是由 Android SDK 提供的，源码位于目录 framework/base/core/java/android.bluetooth/BluetoothDevice.java 中。

文件 BluetoothDevice.java 定义了蓝牙设备属性，代表一个远程蓝牙设备，可以支持 BLE 低功耗设备、BR/EDR 设备或 Dual-mode 类型的设备。通过使用类 BluetoothDevice 可以实现如下功能。

☑ 请求获取远程蓝牙设备的连接。

☑ 查询获取远程蓝牙设备的名称、地址、类和连接状态。

文件 BluetoothDevice.java 的主要实现代码如下所示。

```
static IBluetooth getService() {
            synchronized (BluetoothDevice.class) {
                if (sService == null) {
                    BluetoothAdapter adapter = BluetoothAdapter.getDefaultAdapter();
                    sService = adapter.getBluetoothService(mStateChangeCallback);
                }
            }
            return sService;
    }

    static IBluetoothManagerCallback mStateChangeCallback = new IBluetoothManagerCallback.Stub() {

            public void onBluetoothServiceUp(IBluetooth bluetoothService)
                    throws RemoteException {
                synchronized (BluetoothDevice.class) {
                    sService = bluetoothService;
                }
            }

            public void onBluetoothServiceDown()
                throws RemoteException {
                synchronized (BluetoothDevice.class) {
                    sService = null;
                }
            }
    };
    /*package*/ BluetoothDevice(String address) {
            getService();
            if (!BluetoothAdapter.checkBluetoothAddress(address)) {
                throw new IllegalArgumentException(address + " is not a valid Bluetooth address");
            }

            mAddress = address;
    }
```

```
@Override
public boolean equals(Object o) {
    if (o instanceof BluetoothDevice) {
        return mAddress.equals(((BluetoothDevice)o).getAddress());
    }
    return false;
}

@Override
public int hashCode() {
    return mAddress.hashCode();
}
public static final Parcelable.Creator<BluetoothDevice> CREATOR =
        new Parcelable.Creator<BluetoothDevice>() {
    public BluetoothDevice createFromParcel(Parcel in) {
        return new BluetoothDevice(in.readString());
    }
    public BluetoothDevice[] newArray(int size) {
        return new BluetoothDevice[size];
    }
};
```

在文件 BluetoothDevice.java 中，包含了如下重要的公共方法。

☑ public BluetoothSocket createRfcommSocketToServiceRecord(UUID uuid)

该方法是为了使用带有 listenUsingRfcommWithServiceRecord(String, UUID)方法来进行对等的蓝牙应用而设计的。使用 connect()初始化这个外界连接，并将执行一个已给予 UUID 的 SDP 查找，从而确定连接到哪个通道上。当远程设备将被认证时，在这个端口上的通信会被加密。如果正试图连接蓝牙串口，那么使用众所周知的 SPP UUID 00001101-0000-1000-8000-00805F9B34FB。但是如果正试图连接 Android 设备，那么请生成专有 UUID。参数 uuid 表示查询 RFCOMM 通道的服务记录 UUID。返回值是一个准备好外界连接的 RFCOMM 蓝牙服务端口。

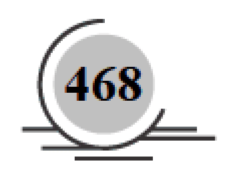
☑ public int describeContents()：描述了包含在 Parcelable's marshalled representation 中的特殊对象的种类。返回值是一个指示被 Parcelabel 所排列的特殊对象类型集合的位屏蔽。

☑ public boolean equals(Object o)

此方法用于比较带有特定目标的常量，如果相等则标示出来。为了保证其相等，o 必须代表相同的对象，该对象作为这个使用类依赖比较的常量。通常约定，该比较需要可复制、相等和可传递。另外，没有对象引用时 null 等于 null。方法 equals()的默认实现是返回 true，当且仅当 this == o。当且仅当 o 是一个作为接收器（使用==操作符来做比较）的精确相同的对象时，这个对象的实现才返回 true 值。子类通常实现 equals(Object)方法，这样才会重视这两个对象的类型和状态。在现实中通常约定，对于 equals(Object)和 hashCode()方法，如果 equals 对于任意两个对象返回真值，那么 hashCode()必须对这些对象返回相同的值。这意味着对象的子类通常都覆盖或者都不覆盖这两个方法。参数 o 表示需要对比常量的对象。对于方法 equals()的返回值来说，如果指定的对象和该对象相等则返回 true，否则返回 false。

☑ public String getAddress()：返回该蓝牙设备的硬件地址，例如，00:11:22:AA:BB:CC，返回值是字符串类型的蓝牙硬件地址。

☑ public BluetoothClass getBluetoothClass()：获取远程设备的蓝牙类，需要 BLUETOOTH 权限。



返回值是蓝牙类对象出错时返回的空值。

- ☑ public int getBondState()：获取远程设备的连接状态，连接状态的可能值有 BOND_NONE、BOND_BONDING 和 BOND_BONDED。返回值是连接状态。
- ☑ public String getName()：获取远程设备的蓝牙用户名。当执行设备扫描时，本地适配器将自动寻找远程名称。该方法只返回来自存储器中该设备的名称，返回值是蓝牙用户名，如果出现问题则返回 null。
- ☑ public int hashCode()：返回该对象的一个整型哈希值，通常约定，如果 equals()对于任意两个对象返回真值，那么 hashCode()必须对这些对象返回相同的值。这意味着对象的子类通常都覆盖或者都不覆盖这两个方法。除非同等对比信息发生改变，否则哈希码不随时间改变而改变。返回值是该对象的哈希值。
- ☑ public String toString()：返回该蓝牙设备的字符串表达式。这是一个蓝牙硬件地址，例如 00:11:22:AA:BB:CC。然而，如果用户明确需要蓝牙硬件地址以防以后 toString()表达式会改变，用户总是需要使用 getAddress()方法。返回值是该蓝牙设备的字符串表达式。
- ☑ public void writeToParcel(Parcel out, int flags)：将类的数据写入外部提供的 Parcel 中。参数 out 表示对象需要被写入的 Parcel，flags 表示和对象需要如何被写入有关的附加标志。可能是 0，或者可能是 1。

由此可见，BluetoothDevice 代表一个远程蓝牙设备，可用于创建一个带有各自设备的 BluetoothDevice 或者查询如名称、地址、类和连接状态等信息。对于蓝牙硬件地址而言，这个类仅仅是一个瘦包装器，类 BluetoothDevice 的对象是不可改变的。对类 BluetoothDevice 上的操作会使用这个用来创建 BluetoothDevice 类的 BluetoothAdapter 类执行在远程蓝牙硬件上。为了获得 BluetoothDevice 类，需要使用 BluetoothAdapter.getRemoteDevice(String)方法去创建一个表示已知 MAC 地址的设备（用户可以通过带有 BluetoothAdapter 类来完成对设备的查找）或者从一个通过 BluetoothAdapter.getBondedDevices()得到返回值的有联系的设备集合来得到该设备。

当搜寻到蓝牙设备后，接下来需要建立蓝牙连接。要想实现两个设备间的 BLE 通信，首先需要建立 GATT 连接，连接 GATT Server。在连接 GATT Server 时需要调用类 BluetoothDevice 中的 connectGatt()函数，此函数的具体实现代码如下所示。

```
    public BluetoothGatt connectGatt(Context context, boolean autoConnect,
                                     BluetoothGattCallback callback, int transport) {
        BluetoothAdapter adapter = BluetoothAdapter.getDefaultAdapter();
        IBluetoothManager managerService = adapter.getBluetoothManager();
        try {
            IBluetoothGatt iGatt = managerService.getBluetoothGatt();
            if (iGatt == null) {
                return null;
            }
            BluetoothGatt gatt = new BluetoothGatt(context, iGatt, this, transport);
            gatt.connect(autoConnect, callback);
            return gatt;
        } catch (RemoteException e) {Log.e(TAG, "", e);}
        return null;
    }
}
```

（3）返回 BluetoothGatt 对象

当通过函数 connectGatt()连接 GATT Server 成功后会返回 BluetoothGatt 对象，这是 GATT profile 的封装，通过这个对象就能进行 GATT Client 端的相关操作。文件 BluetoothGatt.java 的主要实现代码如下所示。

```
public boolean connect() {
    try {
        mService.clientConnect(mClientIf, mDevice.getAddress(),
                                         false, mTransport);
        return true;
    } catch (RemoteException e) {
        Log.e(TAG,"",e);
        return false;
    }
}
public boolean discoverServices() {
    if (DBG) Log.d(TAG, "discoverServices() - device: " + mDevice.getAddress());
    if (mService == null || mClientIf == 0) return false;

    mServices.clear();

    try {
        mService.discoverServices(mClientIf, mDevice.getAddress());
    } catch (RemoteException e) {
        Log.e(TAG,"",e);
        return false;
    }

    return true;
}
public void disconnect() {
    if (DBG) Log.d(TAG, "cancelOpen() - device: " + mDevice.getAddress());
    if (mService == null || mClientIf == 0) return;

    try {
        mService.clientDisconnect(mClientIf, mDevice.getAddress());
    } catch (RemoteException e) {
        Log.e(TAG,"",e);
    }
}
public void close() {
    if (DBG) Log.d(TAG, "close()");

    unregisterApp();
    mConnState = CONN_STATE_CLOSED;
}
public boolean readCharacteristic(BluetoothGattCharacteristic characteristic) {
    if ((characteristic.getProperties() &
            BluetoothGattCharacteristic.PROPERTY_READ) == 0) return false;
```

```
        if (VDBG) Log.d(TAG, "readCharacteristic() - uuid: " + characteristic.getUuid());
        if (mService == null || mClientIf == 0) return false;

        BluetoothGattService service = characteristic.getService();
        if (service == null) return false;

        BluetoothDevice device = service.getDevice();
        if (device == null) return false;

        synchronized(mDeviceBusy) {
            if (mDeviceBusy) return false;
            mDeviceBusy = true;
        }

        try {
            mService.readCharacteristic(mClientIf, device.getAddress(),
                service.getType(), service.getInstanceId(),
                new ParcelUuid(service.getUuid()), characteristic.getInstanceId(),
                new ParcelUuid(characteristic.getUuid()), AUTHENTICATION_NONE);
        } catch (RemoteException e) {
            Log.e(TAG,"",e);
            mDeviceBusy = false;
            return false;
        }

        return true;
    }
    public boolean setCharacteristicNotification(BluetoothGattCharacteristic characteristic,
                                                 boolean enable) {
        if (DBG) Log.d(TAG, "setCharacteristicNotification() - uuid: " + characteristic.getUuid()
                          + " enable: " + enable);
        if (mService == null || mClientIf == 0) return false;

        BluetoothGattService service = characteristic.getService();
        if (service == null) return false;

        BluetoothDevice device = service.getDevice();
        if (device == null) return false;

        try {
            mService.registerForNotification(mClientIf, device.getAddress(),
                service.getType(), service.getInstanceId(),
                new ParcelUuid(service.getUuid()), characteristic.getInstanceId(),
                new ParcelUuid(characteristic.getUuid()),
                enable);
        } catch (RemoteException e) {
            Log.e(TAG,"",e);
            return false;
        }

        return true;
```

```
}
public List<BluetoothGattService> getServices() {
    List<BluetoothGattService> result =
            new ArrayList<BluetoothGattService>();

    for (BluetoothGattService service : mServices) {
        if (service.getDevice().equals(mDevice)) {
            result.add(service);
        }
    }

    return result;
}
```

由此可见，在文件 BluetoothGatt.java 中定义了如下常用的连接操作函数。

☑ connect()：连接远程设备。
☑ discoverServices()：搜索连接设备所支持的 Service。
☑ disconnect()：断开与远程设备的 GATT 连接。
☑ close()：关闭 GATT Client 端。
☑ readCharacteristic(characteristic)：读取指定的 characteristic。
☑ setCharacteristicNotification(characteristic,enabled)：设置当指定 characteristic 值变化时，发出通知。
☑ getServices()：获取远程设备所支持的 Services。

在此需要注意某些函数调用之间存在先后关系。例如，首先需要连接上才能进行 discoverServices。另外，一些函数调用是异步的，需要得到的值不会立即返回，而会在 BluetoothGattCallback 的回调函数中返回。例如，discoverServices 与 onServicesDiscovered 回调，readCharacteristic 与 onCharacteristicRead 回调，setCharacteristicNotification 与 onCharacteristicChanged 回调等。

（4）传递状态和结果

在连接过程中，文件 BluetoothGattCallback.java 用于传递一些连接状态及结果。类 BluetoothGattCallback 返回的是中央状态和周边提供的数据，其中 BluetoothGattServer 作为周边来提供数据，BluetoothGatt 作为中央来使用和处理数据。文件 BluetoothGattCallback.java 的具体实现代码如下所示。

```
public abstract class BluetoothGattCallback {
    public void onConnectionStateChange(BluetoothGatt gatt, int status,
                                        int newState) {
    }
    public void onServicesDiscovered(BluetoothGatt gatt, int status) {
    }
    public void onCharacteristicRead(BluetoothGatt gatt, BluetoothGattCharacteristic characteristic,
                                     int status) {
    }
    public void onCharacteristicWrite(BluetoothGatt gatt,
                                      BluetoothGattCharacteristic characteristic, int status) {
    }
    public void onCharacteristicChanged(BluetoothGatt gatt,
                                        BluetoothGattCharacteristic characteristic) {
```

```
    }
    public void onDescriptorRead(BluetoothGatt gatt, BluetoothGattDescriptor descriptor,
                                    int status) {
    }
    public void onDescriptorWrite(BluetoothGatt gatt, BluetoothGattDescriptor descriptor,
                                    int status) {
    }
    public void onReliableWriteCompleted(BluetoothGatt gatt, int status) {
    }
    public void onReadRemoteRssi(BluetoothGatt gatt, int rssi, int status) {
    }
    public void onMtuChanged(BluetoothGatt gatt, int mtu, int status) {
    }
}
```

（5）实现客户端的低功耗蓝牙规范

在 Broadcom（博通）公司提供的源码中，文件 BleClientProfile.java 的功能是实现客户端的低功耗蓝牙规范。在应用中要想访问远程设备中的低功耗蓝牙规范，就必须继承于类 BleClientProfile，并且需要提供要访问规范的必需参数和服务标识。通过 BleClientProfile 的派生类可以发起一个远程设备的连接，并且一个 BleClientProfile 类可能会包含多个 BleClientService 对象的实例。文件 BleClientProfile.java 的具体实现代码如下所示。

```
//下面是构造方法，功能是给当前规范的 UUID 和客户端应用上下文创建一个 BleClientProfile
    public BleClientProfile(Context context, BleGattID profileUuid)
    {
        Log.d(TAG, "new profile" + profileUuid.toString());

        this.mContext = context;
        this.mAppUuid = profileUuid;

        this.mConnectedDevices = new ArrayList<BluetoothDevice>();
        this.mConnectingDevices = new ArrayList<BluetoothDevice>();
        this.mDisconnectingDevices = new ArrayList<BluetoothDevice>();

        this.mClientIDToDeviceMap = new HashMap<Integer, BluetoothDevice>();
        this.mDeviceToClientIDMap = new HashMap<BluetoothDevice, Integer>();

        this.mCallback = new BleClientCallback();
        this.mSvcConn = new GattServiceConnection(context);
    }

    /**
     * 初始化 BleClientProfile 对象
     */
    public void init(ArrayList<BleClientService> requiredServices,
            ArrayList<BleClientService> optionalServices)
    {
        Log.d(TAG, "init (" + this.mAppUuid + ")");

        this.mRequiredServices = requiredServices;
```

```
        this.mOptionalServices = optionalServices;

        IBinder b = ServiceManager.getService(BleConstants.BLUETOOTH_LE_SERVICE);
        if (b == null) {
            throw new RuntimeException("Bluetooth Low Energy service not available");
        }
        this.mSvcConn.onServiceConnected(null, b);
    }

    /**
     * 清除和此规范有关的资源
     */
    public synchronized void finish()
    {
        if (this.mSvcConn != null) {
            this.mContext.unbindService(this.mSvcConn);
            this.mSvcConn = null;
        }
    }

    @Override

    /**
     *返回此规范是否已经成功注册到蓝牙协议栈中
     * @see {@link #registerProfile()}
     */
    public boolean isProfileRegistered()
    {
        Log.d(TAG, "isProfileRegistered (" + this.mAppUuid + ")");
        return this.mClientIf != BleConstants.GATT_SERVICE_PRIMARY;
    }

    /**
     * 注册规范到蓝牙协议栈
     */
    public int registerProfile()
    {
        int ret = BleConstants.GATT_SUCCESS;
        Log.d(TAG, "registerProfile (" + this.mAppUuid + ")");

        if (this.mClientIf == BleConstants.GATT_SERVICE_PRIMARY)
        {
            try
            {
                this.mService.registerApp(this.mAppUuid, this.mCallback);
            } catch (RemoteException e) {
                Log.e(TAG, e.toString());
                ret = BleConstants.SERVICE_UNAVAILABLE;
            }
        }
```

```
        return ret;
    }

    /**
     *注销蓝牙协议栈中的规范
     */
    public void deregisterProfile()
    {
        Log.d(TAG, "deregisterProfile (" + this.mAppUuid + ")");

        if (this.mClientIf != BleConstants.GATT_SERVICE_PRIMARY)
            try {
                this.mService.unregisterApp(this.mClientIf);
            } catch (RemoteException e) {
                Log.e(TAG, "deregisterProfile() - " + e.toString());
            }
    }

    /**
     * 设置一个活跃连接设备的加密等级
     */
    public void setEncryption(BluetoothDevice device, byte action)
    {
        try
        {
            this.mService.setEncryption(device.getAddress(), action);
        } catch (RemoteException e) {
            Log.e(TAG, e.toString());
        }
    }

    /**
     * 当请求后台连接时，定义本地设备扫描远程低功耗设备的强度
     */
    public void setScanParameters(int scanInterval, int scanWindow)
    {
        try
        {
            this.mService.setScanParameters(scanInterval, scanWindow);
        } catch (RemoteException e) {
            Log.e(TAG, e.toString());
        }
    }

    /**
     * 建立一个到远程设备的 GATT 连接
     */
    public int connect(BluetoothDevice device)
    {
        Log.d(TAG, "connect (" + this.mAppUuid + ")" + device.getAddress());
```

```
        int ret = BleConstants.GATT_SUCCESS;

        synchronized (this.mConnectingDevices) {
            this.mConnectingDevices.add(device);
        }

        synchronized (this.mDisconnectingDevices) {
            this.mDisconnectingDevices.remove(device);
        }
        try
        {
            this.mService.open(this.mClientIf, device.getAddress(), true);
        } catch (RemoteException e) {
            Log.e(TAG, e.toString());
            ret = BleConstants.GATT_ERROR;
        }

        return ret;
    }

    /**
     * 准备一个到远程蓝牙设备的后台连接
     */
    public int connectBackground(BluetoothDevice device)
    {
        Log.d(TAG,
                "connectBackground (" + this.mAppUuid + ")" + device.getAddress());

        int ret = BleConstants.GATT_SUCCESS;

        synchronized (this.mConnectingDevices) {
            this.mConnectingDevices.add(device);
        }

        synchronized (this.mDisconnectingDevices) {
            this.mDisconnectingDevices.remove(device);
        }
        try
        {
            this.mService.open(this.mClientIf, device.getAddress(), false);
        } catch (RemoteException e) {
            Log.e(TAG, e.toString());
            ret = BleConstants.GATT_ERROR;
        }

        return ret;
    }

    /**
     * 停止监听远程蓝牙设备试图发起的连接
     */
```

```
public int cancelBackgroundConnection(BluetoothDevice device)
{
    Log.d(TAG, "cancelBackgroundConnection (" + this.mAppUuid
            + ") - device " + device.getAddress());

    int ret = BleConstants.GATT_SUCCESS;
    try
    {
        this.mService.close(this.mClientIf, device.getAddress(), 0, false);
    } catch (RemoteException e) {
        Log.e(TAG, e.toString());
        ret = BleConstants.GATT_ERROR;
    }

    return ret;
}

/**
 *断开一个到远程设备的 GATT 连接
 */
public int disconnect(BluetoothDevice device)
{
    Log.d(TAG,
            "disconnect (" + this.mAppUuid + ") - device " + device.getAddress());

    synchronized (this.mDisconnectingDevices) {
        this.mDisconnectingDevices.add(device);
    }

    int ret = BleConstants.GATT_SUCCESS;
    try
    {
        this.mService.close(this.mClientIf,
                device.getAddress(),
                ((Integer) this.mDeviceToClientIDMap.get(device)).intValue(),
                true);
    } catch (RemoteException e) {
        Log.e(TAG, e.toString());
        ret = BleConstants.GATT_ERROR;
    }
    return ret;
}

/**
 * 刷新当前客户端的规范
 */
public int refresh(BluetoothDevice device)
{
    Log.d(TAG,
            "refresh (" + this.mAppUuid + ") - address = " + device.getAddress());
```

```
        if (isDeviceDisconnecting(device)) {
            Log.d(TAG, "refresh (" + this.mAppUuid
                    + ") - Device unavailable!");
            return BleConstants.GATT_ERROR;
        }

        this.mRequiredServices.get(BleConstants.GATT_SERVICE_PRIMARY).refresh(device);

        return BleConstants.GATT_SUCCESS;
    }

    /**
     * 刷新当前规范包含的特定服务
     */
    public int refreshService(BluetoothDevice device, BleClientService service)
    {
        Log.d(TAG, "refreshService (" + this.mAppUuid + ") address = s "
                + device.getAddress() + "service = " + service.getServiceId());

        return 0;
    }

    /**
     * 在已经连接的设备列表中查找指定蓝牙设备的地址
     */
    public BluetoothDevice findConnectedDevice(String address)
    {
        BluetoothDevice ret = null;
        synchronized (this.mConnectedDevices) {
            for (int i = 0; i != this.mConnectedDevices.size(); i++) {
                BluetoothDevice d = (BluetoothDevice) this.mConnectedDevices.get(i);
                if (address.equalsIgnoreCase(d.getAddress())) {
                    ret = d;
                    break;
                }
            }
        }
        return ret;
    }

    /**
     * 返回当前连接和等待连接中的所有远程设备集合
     */
    public BluetoothDevice[] getPendingConnections()
    {
        return (BluetoothDevice[]) this.mConnectingDevices.toArray(new BluetoothDevice[0]);
    }

    /**
     *设置一个蓝牙设备地址，在等待连接设备列表中查找一个远程设备
     */
```

```
public BluetoothDevice findDeviceWaitingForConnection(String address)
{
    BluetoothDevice ret = null;
    synchronized (this.mConnectingDevices) {
        for (int i = 0; i < this.mConnectingDevices.size(); i++) {
            BluetoothDevice d = (BluetoothDevice) this.mConnectingDevices.get(i);
            if (address.equalsIgnoreCase(d.getAddress())) {
                ret = d;
                break;
            }
        }
    }
    return ret;
}
```

（6）创建一个代表客户端角色设备上的低功耗蓝牙服务派生类

在 Broadcom 公司提供的源码中，文件 BleClientService.java 的功能是定义一个派生类，此派生类代表了客户端角色设备上的低功耗蓝牙服务。通过这个派生类可以允许应用程序读写低功耗蓝牙服务的特征，并在特征改变时注册通知。文件 BleClientService.java 的主要实现代码如下所示。

```
//定义代表客户端的低功耗服务
public abstract class BleClientService
{
    private static String TAG = "BleClientService";

    private BleClientProfile mProfile = null;
    private BleGattID mServiceId = null;
    private HashMap<BluetoothDevice, ArrayList<ServiceData>> mdeviceToDataMap =
            new HashMap<BluetoothDevice, ArrayList<ServiceData>>();
    private BleCharacteristicDataCallback mCallback =
            new BleCharacteristicDataCallback();
    private boolean mReadDescriptors = true;

    /**
     *创建一个新的低功耗蓝牙服务的 UUID
     *
     * @param serviceId
     */
    public BleClientService(BleGattID serviceId)
    {
        mServiceId = serviceId;
        if (mServiceId.getServiceType() == BleConstants.GATT_UNDEFINED)
            mServiceId.setServiceType(BleConstants.GATT_SERVICE_PRIMARY);
    }

    /**
     * 返回服务的 UUID
     */
    public BleGattID getServiceId()
    {
```

```
        return mServiceId;
    }

    /**
     *写操作远程设备上的一个特性
     */
    public int writeCharacteristic(BluetoothDevice remoteDevice, int instanceId,
            BleCharacteristic characteristic)
    {
        Log.d(TAG, "writeCharacteristic");

        int ret = BleConstants.GATT_SUCCESS;
        int connID = BleConstants.GATT_INVALID_CONN_ID;

        if ((connID = mProfile.getConnIdForDevice(remoteDevice)) == BleConstants.GATT_INVALID_CONN_ID) {
            return BleConstants.GATT_INVALID_CONN_ID;
        }
        ServiceData s = getServiceData(remoteDevice, instanceId);

        if (s == null) {
            return ret;
        }
        s.writeIndex = s.characteristics.indexOf(characteristic);

        if ((s.characteristics != null) && (s.writeIndex >= BleConstants.GATT_SERVICE_PRIMARY)) {
            Log.d(TAG, "writeCharacteristic found characteristic in array:");
            Log.d(TAG,
                    "Service = [instanceID = " + instanceId + " svcid = "
                            + mServiceId.toString() + " serviceType = "
                            + mServiceId.getServiceType());
            Log.d(TAG, "CharID = [instanceID = " + characteristic.getInstanceID()
                    + " svcid = " + characteristic.getID().toString());
            BleGattID svcId = new BleGattID(instanceId, mServiceId.getUuid(),
                    mServiceId.getServiceType());
            BleGattID cID = characteristic.getID();
            BluetoothGattCharID charID = new BluetoothGattCharID(svcId, cID);
            try
            {
                if (characteristic.isDirty()) {
                    if (characteristic.getWriteType() == BleConstants.GATT_SUCCESS)
                        characteristic.setWriteType(2);
                    characteristic.setDirty(false);
                    mProfile.getGattService().writeCharValue(connID, charID,
                            characteristic.getWriteType(), characteristic.getAuthReq(),
                            characteristic.getValue());
                }
                else if (!characteristic.getDirtyDescQueue().isEmpty()) {
                    ArrayList<BleDescriptor> descList =
                            characteristic.getDirtyDescQueue();
                    BleDescriptor descObj = descList.get(0);
```

```
                    Log.d(TAG, "writeCharacteristic - descriptor = "
                            + descObj.getID().toString());
                    if (descObj.isDirty()) {
                        BluetoothGattCharDescrID descID = new BluetoothGattCharDescrID(
                                svcId, cID, descObj.getID());
                        descObj.setDirty(false);
                        mProfile.getGattService().writeCharDescrValue(connID,
                                descID, descObj.getWriteType(), descObj.getAuthReq(),
                                descObj.getValue());
                    }

                }
                else
                {
                    onWriteCharacteristicComplete(0, remoteDevice, characteristic);
                }
            } catch (RemoteException e) {
                ret = BleConstants.GATT_ERROR;
            }
        } else {
            onWriteCharacteristicComplete(0, remoteDevice, characteristic);
        }
        return ret;
    }
```

（7）定义服务器端的角色的低功耗规范

在 Broadcom 公司提供的源码中，文件 BleServerProfile.java 的功能是定义了服务器端的角色的低功耗规范，在创建一个新的低功耗规范之前，需要先继承于这个类，并提供标识要访问规范所必需的参数和服务。通常来说，一个 BleServerProfile 派生的类包含一个或多个 BleServerService 对象。在 BleServerProfile 派生的类中，包含低功耗规范中定义服务的 BleServerService 对象的集合。文件 BleServerProfile.java 的主要实现代码如下所示。

```
public abstract class BleServerProfile
{
    private static final boolean D = true;
    private static final String TAG = "BleServerProfile";
    private Context mCtxt = null;
    private BleGattID mAppid;
    ArrayList<BleServerService> mServiceArr = null;
    private HashMap<String, Integer> mConnMap = null;
    private HashMap<Integer, Integer> mMtuMap = null;
    private IBluetoothGatt mService;
    private int mSvcCreated = 0;
    private int mSvcStarted = 0;
    private byte mAppHandle = -1;
    private int mProfileStatus = 2;
    private GattServiceConnection mSvcConn;

    public BleServerProfile(Context ctxt, BleGattID appId,
            ArrayList<BleServerService> serviceArr)
```

```
{
    mAppid = appId;
    mCtxt = ctxt;
    mServiceArr = serviceArr;
    mConnMap = new HashMap<String, Integer>();
    mMtuMap = new HashMap<Integer, Integer>();
    mSvcConn = new GattServiceConnection(null);
    Intent i = new Intent();
    i.setClassName("com.broadcom.bt.app.system",
            "com.broadcom.bt.app.system.GattService");
    mCtxt.bindService(i, mSvcConn, 1);

    throw new RuntimeException("Not implemented");
}
/*取消和此规范相关的资源*/
public synchronized void finish()
{
    if (mSvcConn != null) {
        mCtxt.unbindService(mSvcConn);
        mSvcConn = null;
    }
}

public void finalize()
{
    finish();
}

byte getAppHandle()
{
    return mAppHandle;
}

HashMap<String, Integer> getConnMap() {
    return mConnMap;
}
/*初始化相关的服务*/
void initProfile()
{
    Log.i("BleServerProfile", "initProfile()");
    try {
        mService.registerServerProfileCallback(mAppid,
                new BleServerProfileCallback(this));
    } catch (Throwable t) {
        Log.e("BleServerProfile", "Unable to start profile", t);
    }
}

void notifyAction(int event)
{
    if ((event == 0) && (++mSvcCreated == mServiceArr.size()))
```

```
        {
            Log.i("BleServerProfile",
                    "All services created successfully. Calling onInitialized");
            onInitialized(true);
        } else if ((event == 4) && (--mSvcCreated == 0))
        {
            Log.i("BleServerProfile",
                    "All services stopped successfully. Calling onStopped");
            onStopped();
        } else if ((event == 2) && (++mSvcStarted == mServiceArr.size()))
        {
            Log.i("BleServerProfile",
                    "All services started successfully. Calling onStarted");
            onStarted(true);
        } else if (event == 1) {
            Log.i("BleServerProfile",
                    "One of the services creation failed. Calling onInitialized");
            mProfileStatus = 2;
            onInitialized(false);
        } else if (event == 3) {
            Log.i("BleServerProfile",
                    "One of the services start failed. Calling onStarted");
            mProfileStatus = 2;
            onStarted(false);
        } else {
            Log.e("BleServerProfile", "Unknown action from a service");
        }
    }
    /*启用和此规范有关的所有服务*/
    public void startProfile()
    {
        Log.i("BleServerProfile", "startProfile()");
        if (mService == null) {
            Log.i("BleServerProfile", "Remote service object is null.. Returning..");
            return;
        }

        for (int i = 0; i < mServiceArr.size(); i++) {
            if (!((BleServerService) mServiceArr.get(i)).isRegistered()) {
                Log.i("BleServerProfile",
                        "One of the services is not registered. Stopping all the services");
                stopProfile();
                return;
            }

            ((BleServerService) mServiceArr.get(i)).startService();
        }
    }
    /*停止和此规范有关的所有服务*/
    public void stopProfile()
    {
```

```
        Log.i("BleServerProfile", "stopProfile()");
        for (int i = 0; i < mServiceArr.size(); i++)
            ((BleServerService) mServiceArr.get(i)).stopService();
    }
    /*注销所有相关的服务*/
    public void finishProfile()
    {
        Log.i("BleServerProfile", "finishProfile()");
        for (int i = 0; i < mServiceArr.size(); i++) {
            ((BleServerService) mServiceArr.get(i)).deleteService();
        }
        try
        {
            mService.unregisterServerProfileCallback(mAppHandle);
        } catch (Throwable t) {
            Log.e("BleServerProfile", "Unable to stop profile", t);
            return;
        }
    }
    /*为连接设置最大传输单元*/
    public void setMtuSize(int connId, int mtuSize)
    {
        Log.i("BleServerProfile", "setMtuSize");
        mMtuMap.put(Integer.valueOf(connId), Integer.valueOf(mtuSize));
    }
    /*为一个活跃的连接设置需要的加密等级*/
    public void setEncryption(String bdaddr, byte action)
    {
        try
        {
            mService.setEncryption(bdaddr, action);
        } catch (Throwable t) {
            Log.e("BleServerProfile", "Unable to set encryption for connection", t);
        }
    }
    /*当已经请求一个后台连接时，定义本地设备扫描远程低功耗设备的强度*/
    public void setScanParameters(int scanInterval, int scanWindow)
    {
        try
        {
            mService.setScanParameters(scanInterval, scanWindow);
        } catch (Throwable t) {
            Log.e("BleServerProfile", "Unable to set scan parameters", t);
        }
    }
    /*打开一个外设 GAP 客户端的连接*/
    public void open(String bdaddr, boolean isDirect)
    {
        try
        {
            mService.GATTServer_Open(mAppHandle, bdaddr, isDirect);
```

```
        } catch (Throwable t) {
            Log.e("BleServerProfile", "Unable to open Gatt connection", t);
        }
    }
    /*取消一个正在进行中对外设 GATT 客户端的打开操作*/
    public void cancelOpen(String bdaddr, boolean isDirect)
    {
        try
        {
            mService.GATTServer_CancelOpen(mAppHandle, bdaddr, isDirect);
        } catch (Throwable t) {
            Log.e("BleServerProfile", "Unable to open Gatt connection", t);
            return;
        }
    }
    /*关闭一个到远程低功耗规范客户端的连接*/
    public void close(String bdaddr)
    {
        try
        {
            mService.GATTServer_Close(((Integer) mConnMap.get(bdaddr))
                    .intValue());
        } catch (Throwable t) {
            Log.e("BleServerProfile", "Unable to open Gatt connection", t);
            return;
        }
    }
```

（8）创建低功耗服务

在 Broadcom 公司提供的源码中，文件 BleServerService.java 的功能是创建一个低功耗服务，这是服务器端角色上的低功耗规范的一部分。在 BleServerService 的派生类中包含了一个或多个 BleCharacteristic 对象。在应用程序中，需要重写类 BleServerService 来实现一个服务。文件 BleServerService.java 的主要实现代码如下所示。

```
public abstract class BleServerService
{
    private final String TAG = "BleServerService";

    private HashMap<Integer, BleCharacteristic> mCharHdlMap = null;
    private HashMap<Integer, BleServerService> mServiceHdlMap = null;
    private HashMap<Integer, AttributeRequestInfo> mAttrReqMap = null;

    private ArrayList<BleCharacteristic> mCharQueue = null;
    private ArrayList<BleDescriptor> mDirtyDescQueue = null;
    private BleGattID mServiceId;
    private BleGattID mAppUuid;
    private BleServerProfile mProfileHandle;
    private IBluetoothGatt mService;
    private int mSvcHandle = -1;
    private byte mSupTransport;
```

```
private BleServiceCallback mGattServiceCallback;
private boolean isServiceAvailable = false;
private int mSvcInstance = 0;

private boolean isPrimary = false;
private int mNumHandles;
private final int CHAR_ADDED = 0;
private final int CHAR_DESC_ADDED = 1;
private final int ATTRIBUTE_WRITE = 2;
private final int ATTRIBUTE_READ = 3;
private final int HDL_VAL_INDICATION = 4;
private final int HDL_VAL_NOTIFICATION = 5;
private final int MTU_EXCHANGE = 6;
private final int EXECUTE_WRITE = 7;

private Handler mHandler = new Handler()
{
    public void handleMessage(Message msg)
    {
    }

};

int getConnId(String address)
{
    if (this.mProfileHandle == null)
        return -1;
    HashMap connMap = this.mProfileHandle.getConnMap();
    return ((Integer) connMap.get(address)).intValue();
}
/*构造函数，使用给定的 ID 构造一个低功耗服务*/
public BleServerService(BleGattID serviceId, int numHandles)
{
    /**
     * TODO: implement
     */
    this.mServiceId = serviceId;
    this.mNumHandles = numHandles;
    this.mSupTransport = 2;
    this.mGattServiceCallback = new BleServiceCallback(this);
    this.mCharHdlMap = new HashMap();
    this.mServiceHdlMap = new HashMap();
    this.mCharQueue = new ArrayList();
    this.mAttrReqMap = new HashMap();

    if (this.mServiceId.getServiceType() == -1)
        this.mServiceId.setServiceType(0);
    throw new RuntimeException("not implemented");

}
/*构造函数，使用给定的 ID 构造一个新的低功耗服务*/
```

```
public BleServerService(BleGattID serviceId, byte supTransport, int numHandles)
{
    this.mServiceId = serviceId;
    this.mNumHandles = numHandles;
    this.mSupTransport = supTransport;
    this.mGattServiceCallback = new BleServiceCallback(this);

    this.mCharHdlMap = new HashMap();
    this.mCharQueue = new ArrayList();
    this.mServiceHdlMap = new HashMap();
    this.mAttrReqMap = new HashMap();

    if (this.mServiceId.getServiceType() == -1)
        this.mServiceId.setServiceType(0);
    throw new RuntimeException("not implemented");
}
/*初始化服务*/
protected void initService()
{
    if (this.mService != null)
        try {
            this.mService.registerServerServiceCallback(this.mServiceId,
                    this.mAppUuid, this.mGattServiceCallback);
        } catch (Throwable t) {
            Log.e("BleServerService", "initService", t);
        }
}
/*注册服务到蓝牙协议栈*/
public void createService()
{
    if (this.mService != null)
        try {
            this.mService.GATTServer_CreateService(this.mProfileHandle.getAppHandle(),
                    this.mServiceId, this.mNumHandles);
        } catch (Throwable t)
        {
            Log.e("BleServerService", "createService", t);
        }
}
/*从蓝牙协议栈注销服务*/
public void deleteService()
{
    if (this.mService != null)
        try {
            this.mService.GATTServer_DeleteService(this.mSvcHandle);
        } catch (Throwable t) {
            Log.e("BleServerService", "deleteService ", t);
        }
}
/*启用服务*/
public void startService()
```

```
{
    if (this.mService != null)
        try {
            this.mService.GATTServer_StartService(this.mSvcHandle, this.mSupTransport);
        } catch (Throwable t) {
            Log.e("BleServerService", "startService ", t);
        }
}
/*停止服务*/
public void stopService()
{
    if (this.mService != null) {
        this.mProfileHandle.notifyAction(4);
        try {
            this.mService.unregisterServerServiceCallback(this.mSvcHandle);
            this.mService.GATTServer_StopService(this.mSvcHandle);
        } catch (Throwable t) {
            Log.e("BleServerService", "stopService ", t);
        }
    }
}
/*为此服务添加一个包含的服务*/
public void addIncludedService(BleServerService service)
{
    if (this.mService != null)
        try {
            if (service.isRegistered()) {
                this.mServiceHdlMap.put(Integer.valueOf(service.getServiceHandle()),
                        service);
                this.mService.GATTServer_AddIncludedService(this.mSvcHandle,
                        service.getServiceHandle());
            }
            else {
                Log.i("BleServerService",
                        "addIncludedService: Service to be included is not registered.");
            }
        } catch (Throwable t) {
            Log.e("BleServerService", "addIncludedService", t);
        }
}
/*更新一个特性或描述符*/
public void updateCharacteristic(BleCharacteristic charObj)
{
    addCharacteristic(charObj);
}
/*当客户端已经请求读或写一个本地特性属性后发送一个响应*/
public void sendResponse(String address, int transId, byte[] data, int statusCode)
{
    Log.d("BleServerService", "sendResponse() address = " + address + ", transId = "
            + transId + ",statusCode = " + statusCode);
```

```
        if (this.mService == null) {
            Log.e("BleServerService", "sendResponse(): error. GattService not available");
            return;
        }

        AttributeRequestInfo attrInfo = (AttributeRequestInfo) this.mAttrReqMap
                .remove(Integer.valueOf(transId));
        if (attrInfo == null)
        {
            Log.e("BleServerService",
                    "sendResponse() error. attrInfo not found with transId " + transId);
            return;
        }

        byte[] dataToSend = null;
        if (attrInfo.mOffset == 0) {
            dataToSend = data;
        } else {
            dataToSend = new byte[data.length - attrInfo.mOffset];
            System.arraycopy(data, attrInfo.mOffset, dataToSend, 0, dataToSend.length);
        }

        try
        {
            this.mService
                    .GATTServer_SendRsp(
                            attrInfo.mConnId,
                            attrInfo.mTransId,
                            (byte) statusCode,
                            attrInfo.mAttrHandle,
                            attrInfo.mOffset,
                            dataToSend,
                            (byte) 0,
                            false);
        } catch (Throwable t)
        {
            Log.e("BleServerService", "sendResponse(): error", t);
        }
    }
```

（9）描述低功耗蓝牙服务的特性

在 Broadcom 公司提供的源码中，文件 BleCharacteristic.java 的功能是描述低功耗蓝牙服务的特性。在特性中包含了描述符、实际值和元数据，提供了表现格式或便于阅读值的描述。文件 BleCharacteristic.java 的主要实现代码如下所示。

```
public class BleCharacteristic extends BleAttribute
        implements Parcelable
{
    private static final String TAG = "BleCharacteristic";
    private HashMap<BleGattID, BleDescriptor> mDescriptorMap = new HashMap<BleGattID, BleDescriptor>();
```

```
private ArrayList<BleDescriptor> mDirtyDescQueue = new ArrayList<BleDescriptor>();
private int mProp;
private int mWriteType;
private byte mAuthReq;
private int mPermission = 0;

/** @hide */
@SuppressWarnings({
            "rawtypes", "unchecked"
})
public static final Parcelable.Creator<BleCharacteristic> CREATOR = new Parcelable.Creator()
{
    @Override
    public BleCharacteristic createFromParcel(Parcel source) {
        return new BleCharacteristic(source);
    }

};

/*获取 GATT 的 ID 值*/
private BleGattID getBleGattId(int handle)
{
    for (Map.Entry<BleGattID, Integer> entry : mHandleMap.entrySet()) {
        if (handle == entry.getValue().intValue()) {
            return entry.getKey();
        }
    }
    return null;
}

/**
 *返回该特征的实例的 ID
 *实例 ID 的 BLE 配置文件和服务用于标识属于一个给定的实例的服务或轮廓的特征
 */
public int getInstanceID()
{
    return mID.getInstanceID();
}

/**
 *指定一个实例 ID 的这一特性
 *
 * @see {@link #getInstanceID()}
 */
public void setInstanceID(int instanceID)
{
    mID.setInstanceId(instanceID);
}

/**
 *根据特性向一个给定的偏移量设置原始值的字节
```

```
 */
public byte setValue(byte[] value, int offset, int len, int handle, int totalsize,
          String address)
{
     int uuid = -1;
     int uuidType = -1;
     Log.e("BleCharacteristic", "#### handle is " + handle + " total size is "
               + totalsize);

     BleGattID gattUuid = getBleGattId(handle);
     if (gattUuid == null) {
          Log.e("BleCharacteristic", "setValue: Invalid handle");
          return BleConstants.GATT_INVALID_HANDLE;
     }

     if (gattUuid.equals(mID)) {
          Log.i("BleCharacteristic", "##Writing a characteristic value..");
          Log.i("BleCharacteristic", "##offset=" + offset + " mMaxLength="
                    + mMaxLength + " totalsize=" + totalsize);
          return setValue(value, offset, len, gattUuid, totalsize, address);
     }
     BleDescriptor descObj = mDescriptorMap.get(gattUuid);
     if (descObj != null) {
          Log.i("BleCharacteristic", "##Writing descriptor value..");
          Log.i("BleCharacteristic",
                    "##offset=" + offset + " mMaxSize=" + descObj.getMaxLength() + " totalsize="
                              + totalsize + "desc uuid =" + descObj.getID());
          if (offset > descObj.getMaxLength())
               return BleConstants.GATT_INVALID_OFFSET;
          if (offset + totalsize > descObj.getMaxLength())
               return BleConstants.GATT_INVALID_ATTR_LEN;
          Log.i("BleCharacteristic", "find the user defined descriptor ");
          return descObj.setValue(value, offset, len, gattUuid, totalsize, address);
     }
     Log.e("BleCharacteristic", "Failed to write the value correctly!!!");
     return -127;
}
```

（10）低功耗描述符

在 Broadcom 公司提供的源码中，文件 BleDescriptor.java 是 BleCharacteristic 的一部分，功能是定义了一个低功耗描述符。文件 BleDescriptor.java 的主要实现代码如下所示。

```
public class BleDescriptor extends BleAttribute
          implements Parcelable
{
     private static final String TAG = "BleDescriptor";
     private BleCharacteristic mCharObj;
     protected HashMap<String, Integer> mClientcfgMap = new HashMap();

     /** @hide */
     @SuppressWarnings({
```

```
            "unchecked", "rawtypes"
})
public static final Parcelable.Creator<BleDescriptor> CREATOR = new Parcelable.Creator()
{
    public BleDescriptor createFromParcel(Parcel source) {
        return new BleDescriptor(source);
    }

    public BleDescriptor[] newArray(int size)
    {
        return new BleDescriptor[size];
    }

};

/**
 *从一个给定的偏移设置原始值的字节的描述符
 *
 * @return {@link BleConstants#GATT_SUCCESS if successful}
 */
@Override
public byte setValue(byte[] value, int offset, int length, BleGattID gattUuid,
        int totalSize, String address)
{
    int uuidType = gattUuid.getUuidType();
    int uuid = -1;
    Log.e("BleDescriptor", "#### UUID type=" + gattUuid.getUuidType());

    if (uuidType == 2) {
        uuid = gattUuid.getUuid16();
        if (uuid == -1) {
            Log.e("BleDescriptor", "setValue: Invalid handle (UUID16 not found)");
            return 1;
        }
    }
    if (uuid == 10500) {
        Log.i("BleDescriptor", "##Writing a Presentation format..");
    } else if (uuid == 10498) {
        Log.i("BleDescriptor", "##Writing a characteristic client config");
        if (totalSize > this.mMaxLength)
            return 13;
        int valueInt = 0;
        for (int i = 0; i < length; i++) {
            int shift = (length - 1 - i) * 8;
            valueInt += ((value[i] & 0xFF) << shift);
        }
        this.mClientcfgMap.put(address, Integer.valueOf(valueInt));
    } else if (gattUuid.equals(this.mID)) {
        Log.i("BleDescriptor", "##Writing a descriptor value..");
        Log.i("BleDescriptor", "##offset=" + offset + " mMaxLength=" + this.mMaxLength
                + " length=" + length);
```

```
            super.setValue(value, offset, length, gattUuid, totalSize, address);
        }
        this.mDirty = true;
        return 0;
    }
}
```

（11）标识低功耗蓝牙规范、服务和特性

在 Broadcom 公司提供的源码中，文件 BleGattID.java 的功能是定义了一个标识低功耗蓝牙规范、服务和特性的类，此类使用 16 位或 128 位的 UUIDs 来标识一个给定的低功耗蓝牙实体，这个实体包含规范、服务和特性。文件 BleGattID.java 的主要实现代码如下所示。

```
/**
 *标识一个蓝牙 GATT 特性或属性
 */
public final class BleGattID extends BluetoothGattID
        implements Parcelable
{
    private static final String BASE_UUID_TPL = "%08x-0000-1000-8000-00805f9b34fb";
    @SuppressWarnings({
            "rawtypes", "unchecked"
    })
    public static final Parcelable.Creator<BleGattID> CREATOR = new Parcelable.Creator() {
        public BleGattID createFromParcel(Parcel source) {
            int instId = source.readInt();
            int type = source.readInt();
            int serviceType = source.readInt();

            if (type == 16) {
                String sUuid = source.readString();
                return new BleGattID(instId, sUuid, serviceType);
            }
            int uuid = source.readInt();
            return new BleGattID(instId, uuid, serviceType);
        }

        public BleGattID[] newArray(int size)
        {
            return new BleGattID[size];
        }

    };
```

（12）为远程蓝牙设备提供额外信息

在 Broadcom 公司提供的源码中，文件 BleAdapter.java 的功能是为远程蓝牙设备提供额外的信息，能够判断远程设备是否是低功耗设备、BR/EDR 传统蓝牙设备或双模设备（同时支持低功耗和传统设备）。文件 BleAdapter.java 的主要实现代码如下所示。

```
/**
 *提供帮助的功能和相关的常数扩展蓝牙功能的低能耗信息
```

```
 */
public class BleAdapter
{
    private static final String TAG = "BleAdapter";
    private static final boolean D = true;

    private static final int API_LEVEL = 5;
    private static IBluetoothGatt mService;
    private GattServiceConnection mSvcConn;
    private Context mContext;

    /**
     * 设置远程 ACTION_FOUND 设备的额外信息
     *
     * @see {@link #DEVICE_TYPE_BREDR}, {@link #DEVICE_TYPE_BLE},
     *        {@link #DEVICE_TYPE_DUMO}
     */
    public static final String EXTRA_DEVICE_TYPE = "android.bluetooth.device.extra.DEVICE_TYPE";

    /**
     * Identifies a remote Bluetooth device as type BR/EDR, not capable of
     * accepting Bluetooth Low Energy connections
     */
    public static final byte DEVICE_TYPE_BREDR = 1;

    /**
     * Designates a remote device as a Bluetooth Low Energy (only) device
     */
    public static final byte DEVICE_TYPE_BLE = 2;
    public static final byte DEVICE_TYPE_DUMO = 3;
    public static final String ACTION_UUID = "android.bluetooth.le.device.action.UUID";

    public static final String EXTRA_UUID = "android.bluetooth.le.device.extra.UUID";
    public static final String EXTRA_DEVICE = "android.bluetooth.le.device.extra.DEVICE";

    private static boolean startService() {
        if (mService != null)
            return true;
        IBinder service = ServiceManager.getService(BleConstants.BLUETOOTH_LE_SERVICE);
        if (service != null)
            mService = IBluetoothGatt.Stub.asInterface(service);
        return mService != null;
    }

    /**
     * 构建一种新的 BleAdapter 对象
     */
    public BleAdapter(Context ctx) {
        this.mContext = ctx;
        if (startService()==false)
```

```
            throw new RuntimeException("failed connecting to service");
        this.init();
    }

    /**
     * 启动远程设备中的蓝牙服务，发现使用{@link #ACTION_UUID}的意图
     */
    public static boolean getRemoteServices(String deviceAddress)
    {
        if (!startService())
            throw new RuntimeException("service not available");
        BluetoothAdapter adapter = BluetoothAdapter.getDefaultAdapter();
        if (adapter == null)
            return false;
        try {
            mService.getUUIDs(deviceAddress);
            return true;
        } catch (RemoteException e) {
            e.printStackTrace();
            if (D)
                Log.e(TAG, "error", e);
        }
        return false;
    }
```

（13）保存和 GATT 相关的常量

在 Broadcom 公司提供的源码中，文件 BleConstants.java 的功能是定义保存各种和 GATT 相关的常量，这些常量用于表示各种实现低功耗功能函数的属性和返回值。文件 BleConstants.java 的主要实现代码如下所示。

```
public abstract class BleConstants
{
    public static final int GATT_UNDEFINED = -1;
    public static final int GATT_SERVICE_CREATION_SUCCESS = 0;
    public static final int GATT_SERVICE_CREATION_FAILED = 1;
    public static final int GATT_SERVICE_START_SUCCESS = 2;
    public static final int GATT_SERVICE_START_FAILED = 3;
    public static final int GATT_SERVICE_STOPPED = 4;
    public static final int SERVICE_UNAVAILABLE = 1;
    public static final int GATT_SERVICE_PRIMARY = 0;
    public static final int GATT_SERVICE_SECONDARY = 1;
    public static final int GATT_SERVER_PROFILE_INITIALIZED = 0;
    public static final int GATT_SERVER_PROFILE_UP = 1;
    public static final int GATT_SERVER_PROFILE_DOWN = 2;
    public static final int GATT_SUCCESS = 0;
    public static final int GATT_INVALID_HANDLE = 1;
    public static final int GATT_READ_NOT_PERMIT = 2;
    public static final int GATT_WRITE_NOT_PERMIT = 3;
    public static final int GATT_INVALID_PDU = 4;
```

```
public static final int GATT_INSUF_AUTHENTICATION = 5;
public static final int GATT_REQ_NOT_SUPPORTED = 6;
public static final int GATT_INVALID_OFFSET = 7;
public static final int GATT_INSUF_AUTHORIZATION = 8;
public static final int GATT_PREPARE_Q_FULL = 9;
public static final int GATT_NOT_FOUND = 10;
public static final int GATT_NOT_LONG = 11;
public static final int GATT_INSUF_KEY_SIZE = 12;
public static final int GATT_INVALID_ATTR_LEN = 13;
public static final int GATT_ERR_UNLIKELY = 14;
public static final int GATT_INSUF_ENCRYPTION = 15;
public static final int GATT_UNSUPPORT_GRP_TYPE = 16;
public static final int GATT_INSUF_RESOURCE = 17;
public static final int GATT_ILLEGAL_PARAMETER = 135;
public static final int GATT_NO_RESOURCES = 128;
public static final int GATT_INTERNAL_ERROR = 129;
public static final int GATT_WRONG_STATE = 130;
public static final int GATT_DB_FULL = 131;
public static final int GATT_BUSY = 132;
public static final int GATT_ERROR = 133;
public static final int GATT_CMD_STARTED = 134;
public static final int GATT_PENDING = 136;
public static final int GATT_AUTH_FAIL = 137;
public static final int GATT_MORE = 138;
public static final int GATT_INVALID_CFG = 139;
public static final byte GATT_AUTH_REQ_NONE = 0;
public static final byte GATT_AUTH_REQ_NO_MITM = 1;
public static final byte GATT_AUTH_REQ_MITM = 2;
public static final byte GATT_AUTH_REQ_SIGNED_NO_MITM = 3;
public static final byte GATT_AUTH_REQ_SIGNED_MITM = 4;
public static final int GATT_PERM_READ = 1;
public static final int GATT_PERM_READ_ENCRYPTED = 2;
public static final int GATT_PERM_READ_ENC_MITM = 4;
public static final int GATT_PERM_WRITE = 16;
public static final int GATT_PERM_WRITE_ENCRYPTED = 32;
public static final int GATT_PERM_WRITE_ENC_MITM = 64;
public static final int GATT_PERM_WRITE_SIGNED = 128;
public static final int GATT_PERM_WRITE_SIGNED_MITM = 256;
public static final byte GATT_CHAR_PROP_BIT_BROADCAST = 1;
public static final byte GATT_CHAR_PROP_BIT_READ = 2;
public static final byte GATT_CHAR_PROP_BIT_WRITE_NR = 4;
public static final byte GATT_CHAR_PROP_BIT_WRITE = 8;
public static final byte GATT_CHAR_PROP_BIT_NOTIFY = 16;
public static final byte GATT_CHAR_PROP_BIT_INDICATE = 32;
public static final byte GATT_CHAR_PROP_BIT_AUTH = 64;
public static final byte GATT_CHAR_PROP_BIT_EXT_PROP = -128;
public static final byte SVC_INF_INVALID = -1;
public static final int GATTC_TYPE_WRITE_NO_RSP = 1;
public static final int GATTC_TYPE_WRITE = 2;
public static final int GATT_FORMAT_RES = 0;
```

```
public static final int GATT_FORMAT_BOOL = 1;
public static final int GATT_FORMAT_2BITS = 2;
public static final int GATT_FORMAT_NIBBLE = 3;
public static final int GATT_FORMAT_UINT8 = 4;
public static final int GATT_FORMAT_UINT12 = 5;
public static final int GATT_FORMAT_UINT16 = 6;
public static final int GATT_FORMAT_UINT24 = 7;
public static final int GATT_FORMAT_UINT32 = 8;
public static final int GATT_FORMAT_UINT48 = 9;
public static final int GATT_FORMAT_UINT64 = 10;
public static final int GATT_FORMAT_UINT128 = 11;
public static final int GATT_FORMAT_SINT8 = 12;
public static final int GATT_FORMAT_SINT12 = 13;
public static final int GATT_FORMAT_SINT16 = 14;
public static final int GATT_FORMAT_SINT24 = 15;
public static final int GATT_FORMAT_SINT32 = 16;
public static final int GATT_FORMAT_SINT48 = 17;
public static final int GATT_FORMAT_SINT64 = 18;
public static final int GATT_FORMAT_SINT128 = 19;
public static final int GATT_FORMAT_FLOAT32 = 20;
public static final int GATT_FORMAT_FLOAT64 = 21;
public static final int GATT_FORMAT_SFLOAT = 22;
public static final int GATT_FORMAT_FLOAT = 23;
public static final int GATT_FORMAT_DUINT16 = 24;
public static final int GATT_FORMAT_UTF8S = 25;
public static final int GATT_FORMAT_UTF16S = 26;
public static final int GATT_FORMAT_STRUCT = 27;
```

到此为止，Broadcom 公司推出的低功耗蓝牙协议栈 BlueDroid 的开发文档和 API 源码分析完毕。因为本书篇幅的限制，只是分析了主要的模块类，其他类的实现代码的功能和原理请读者参阅其源码中的注释说明。

第 13 章　Android 多媒体框架架构详解

从 Android 2.2 版本以后，Android 对多媒体框架进行了很大的调整，抛弃了原来的 OpenCore 框架，改用 StageFright 框架，仅对 OpenCore 中的 omx-component 部分做了引用。和 OpenCore 框架相比，StageFright 框架更加易懂，并且封装也相对简单。在 Android 2.2 及以前版本中，OpenCore 位于 external 目录下，在 Android 2.3 以后，多媒体的功能被放置到 frameworks/base/media 目录下。本章将详细讲解 OpenCore 框架和 StageFright 框架的基本知识，为读者学习本书后面的知识打下基础。

13.1　Android 多媒体系统介绍

在 Android 的多媒体系统中，可以根据需要添加一些第三方插件，这样可以增强多媒体系统的功能。在 Android 系统的本地多媒体引擎上面，是 Android 的多媒体本地框架，而在多媒体本地框架上面是多媒体 JNI 和多媒体的 Java 框架部分。多媒体相关的应用程序通过调用 Android Java 框架层来提供标准的多媒体 API 进行构建。本章将要讲解的 OpenCore 引擎和 StageFright 引擎是 Android 本地框架中定义接口的实现者，上层调用者不知道 Android 下层使用什么多媒体引擎。

Android 多媒体引擎和插件的基本层次结构如图 13-1 所示。

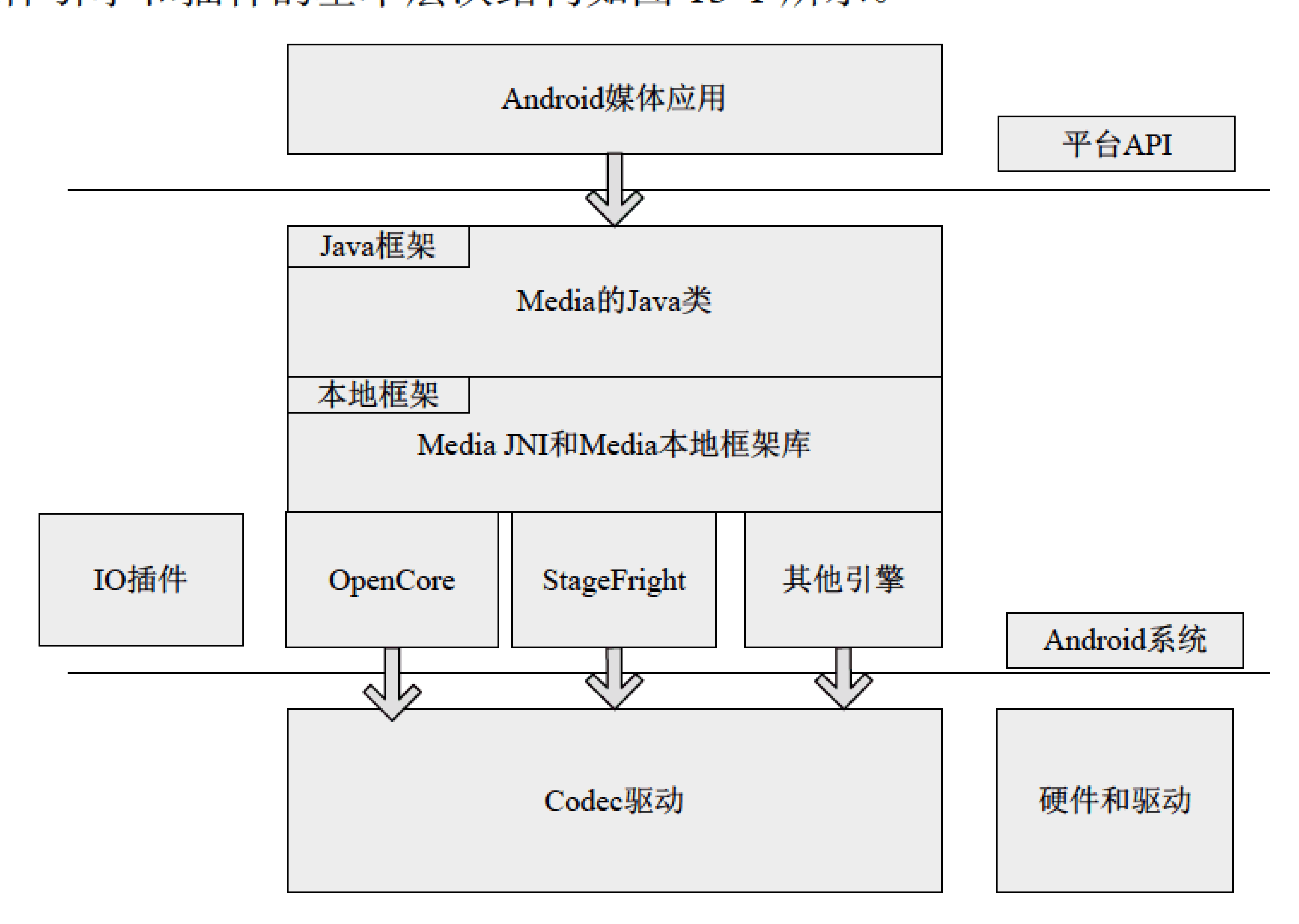


图 13-1　Android 多媒体引擎和插件的基本层次

Android 系统的多媒体框架结构如图 13-2 所示。

从多媒体应用的实现角度来看，多媒体系统主要包含如下两方面的内容。

（1）输入/输出环节：音频、视频纯数据流的输入、输出系统。

（2）中间处理环节：包括文件格式处理和编码/解码环节处理。

图 13-2　Android 系统的多媒体框架结构

假如想要处理一个 MP3 文件，媒体播放器的处理流程是：将一个 MP3 格式的文件作为播放器的输入，将声音从播放器设备输出。在具体实现上，MP3 播放器经过了 MP3 格式文件解析、MP3 码流解码和 PCM 输出播放的过程。整个过程如图 13-3 所示。

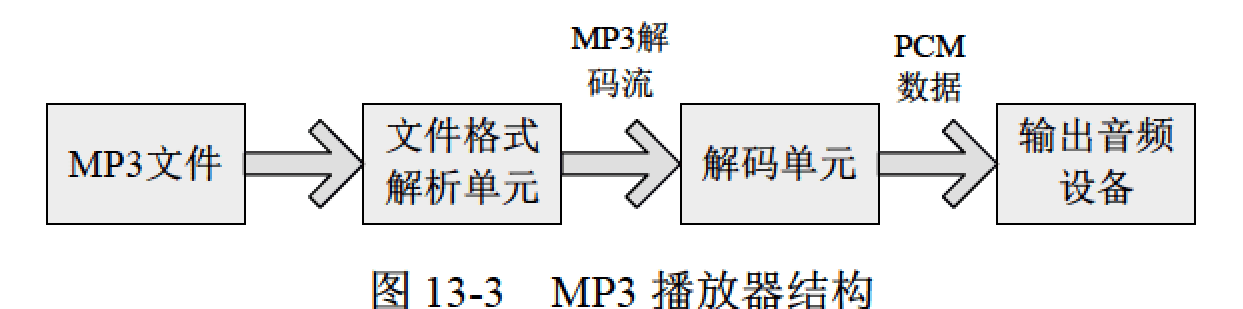


图 13-3　MP3 播放器结构

13.2　OpenMax 框架详解

2006 年，NVIDIA 公司和 Khronos 联合推出 OpenMax，这是多媒体应用程序的框架标准。OpenMax 是无授权费的、跨平台的应用程序接口 API。OpenMax 通过使媒体加速组件能够在开发、集成和编程环节中实现跨多操作系统和处理器硬件平台，提供全面的流媒体编码/解码器和应用程序便携化。

OpenMax 的官方网站地址为 http://www.khronos.org/openmax/。

OpenMax 是一个多媒体应用程序的框架标准。在此标准中集成层的 OpenMax IL 中定义了媒体组件接口，通过这些接口可以在嵌入式器件的流媒体框架中实现对加速编码器和解码器的快速集成。

Android 系统本身没有独立的多媒体系统，而是直接使用了市面中现成的产品，OpenMax IL 便是其中之一。在 Android 结构中，OpenMax IL 通常被当作多媒体引擎插件来使用。Android 最早的多媒体引擎是 OpenCore，后续版本逐渐使用 StageFright 来代替。这两种引擎都可以使用 OpenMax 作为插

件，主要实现编码和解码（Codec）处理。

在 Android 的框架层中定义了由 Android 封装的 OpenMax 接口，此接口和标准的接口类似。但是因为使用的是 C++类型接口，并且使用了 Android 的 Binder IPC 机制，所以处理速度会很快。后续引擎 StageFright 使用了封装的 OpenMax 接口，而早期引擎 OpenCore 并没有使用此接口，而是使用其他形式封装了 OpenMax IL 层接口。

Android 中 OpenMax 的基本层次结构如图 13-4 所示。

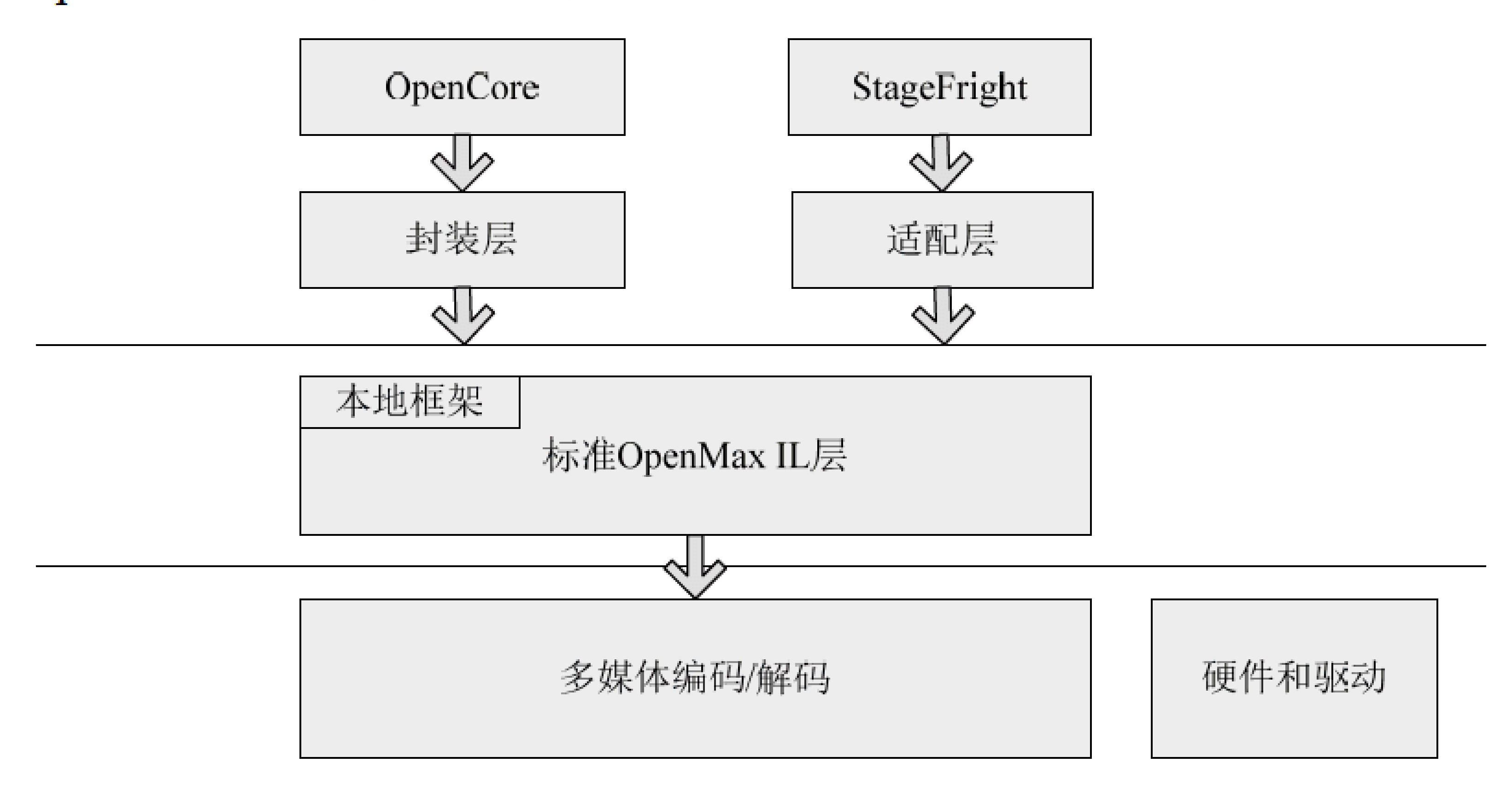


图 13-4　OpenMax 多媒体框架的层次结构

13.2.1　分析 OpenMax 框架构成

在图 13-4 中列出了 OpenMax 多媒体框架的层次结构，下面将详细讲解各个层次结构的基本知识。

1．OpenMax 总体层次结构

OpenMax 分为 3 个层次，从上到下分别是 OpenMax DL（Development Layer，开发层）、OpenMax IL（Integration Layer，集成层）和 OpenMax AL（Application Layer，应用层）。在实际的应用中，OpenMax 的 3 个层次中使用较多的是 OpenMax IL 层，由于操作系统到硬件的差异和多媒体应用的差异，OpenMax 的 DL 和 AL 层使用相对较少。

接下来讲解上述 3 个层次结构的具体说明。

（1）OpenMax DL

在 OpenMax DL 中定义了集音频、视频和图像功能的 API，这样供应商可以在一个新的处理器上实现并优化，然后编码/解码供应商使用该法来编写更广泛的编码/解码器功能。OpenMax DL 可以处理 FFT 和 Filter 等音频信号，也可以实现颜色空间转换和处理原始视频，并且可以实现对诸如 MPEG-4、H.264、MP3、AAC 和 JPEG 等编码/解码器的优化。

（2）OpenMax IL

OpenMax IL 是一种音频、视频和图像编码/解码器，能够实现和多媒体编码/解码器的交互。OpenMax IL 的主要目的是使用特征集合为编码/解码器提供一个系统抽象，解决多个不同媒体系统之间的轻便性问题。

（3）OpenMax AL

OpenMax AL API 在应用程序和多媒体中间件之间提供了一个标准化接口，多媒体中间件提供服务



以实现被期待的 API 功能。OpenMax 具有 3 个层次，如图 13-5 所示。

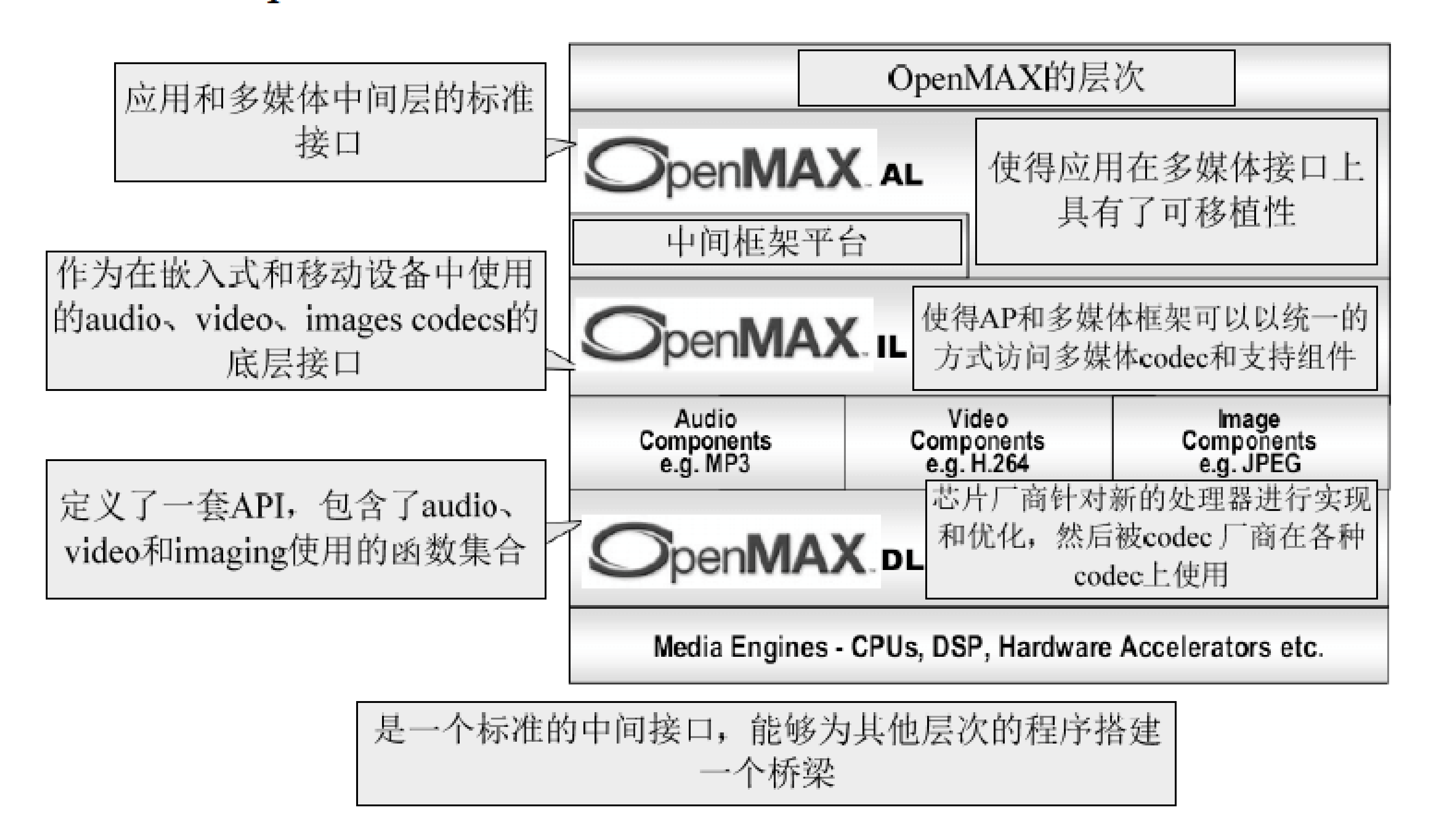


图 13-5　OpenMax 层次

2．OpenMax IL 层的结构

在当前多媒体领域，因为 OpenMax IL 的普及性，实际上已经成为了多媒体框架标准。大多数嵌入式处理器或者多媒体编码/解码器模块的硬件生产者通常都提供了标准的 OpenMax IL 层的软件接口，这样程序员就可以基于此层次的标准化接口进行多媒体程序的开发。

OpenMax IL 的接口层次结构比较科学，既不是硬件编码/解码器的接口，也不是应用程序层的接口，所以可以比较容易地实现标准化。OpenMax IL 的层次结构如图 13-6 所示。

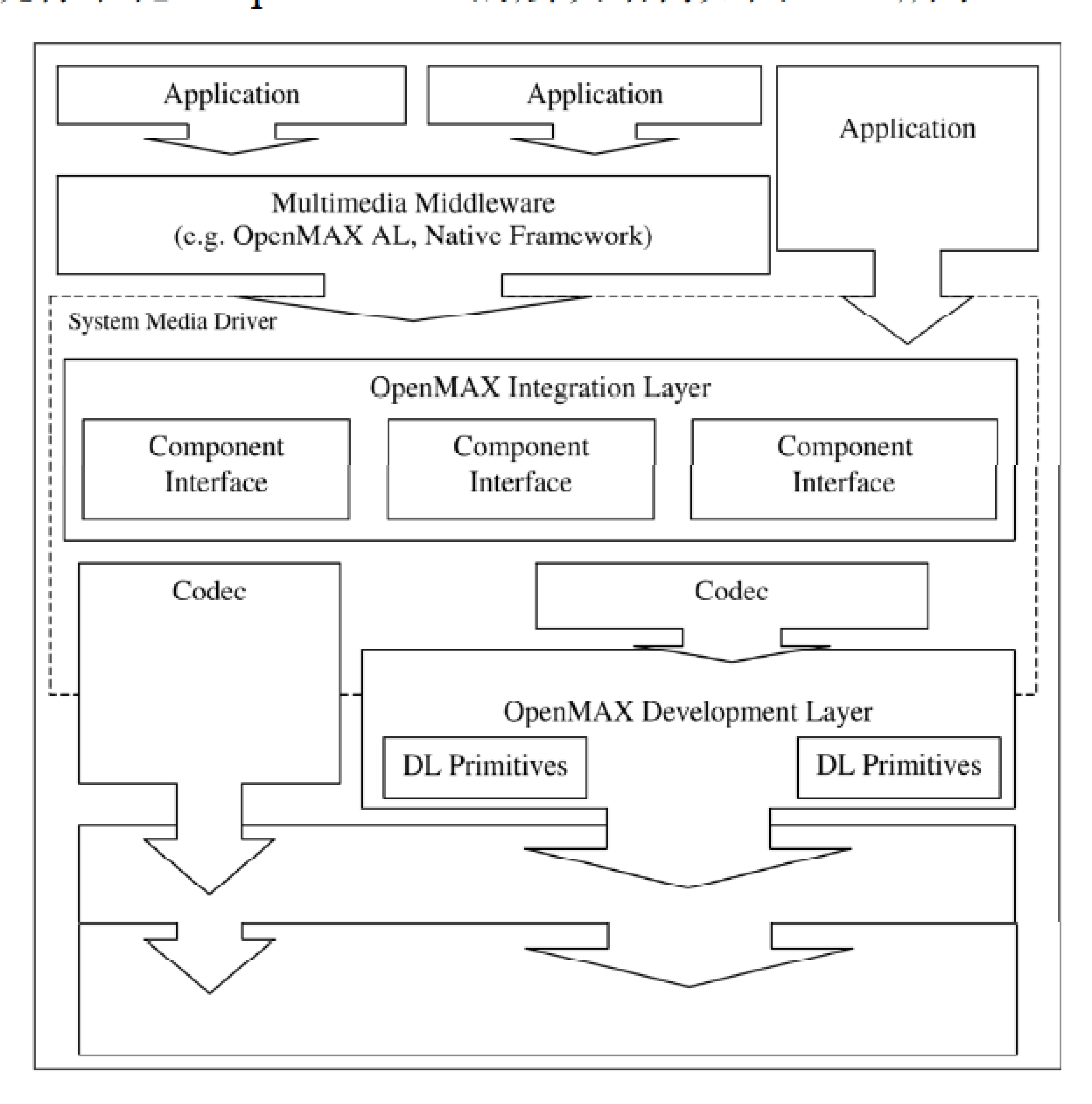


图 13-6　OpenMax IL 的层次结构

在图 13-6 所示的层次结构中，虚线部分中是 OpenMax IL 层的内容，功能是实现了 OpenMax IL 中的各个组件（Component）。对于下层而言，OpenMax IL 既可以调用 OpenMax DL 层的接口，也可

以直接调用各种 Codec 实现。对于上层而言，OpenMax IL 既可以给 OpenMax AL 层等框架层（Middleware）调用，也可以给应用程序直接调用。

OpenMax IL 层中包含的主要内容如下所示。

☑ Client：客户端，OpenMax IL 的调用者。
☑ Component：组件，OpenMax IL 的单元，每一个组件实现一种功能。
☑ Port：端口，组件的输入/输出接口。
☑ Tunneled：隧道化，让两个组件直接连接的方式。

OpenMax IL 层的运作流程如图 13-7 所示。

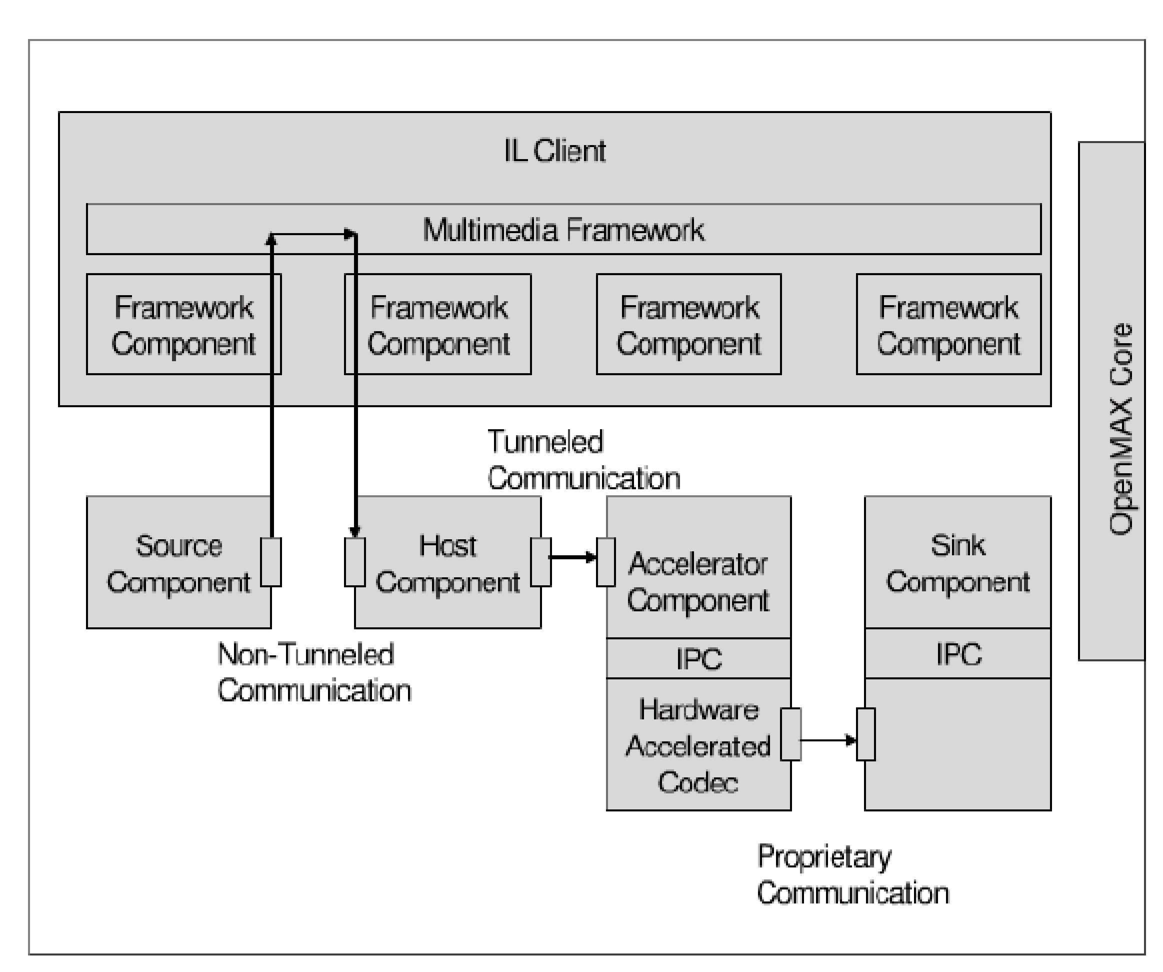


图 13-7　OpenMax IL 层的运作流程

在图 13-7 中，OpenMax IL 层的客户端通过调用如下 4 个 OpenMax IL 组件来实现同一个功能。

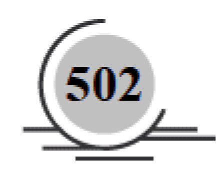
☑ Source 组件：只有一个输出端口。
☑ Host 组件：有一个输入端口和一个输出端口。
☑ Accelerator 组件：具有一个输入端口，调用了硬件的编码/解码器，加速主要体现在此环节。
☑ Sink 组件：Accelerator 组件和 Sink 组件通过私有通信方式在内部进行连接，没有经过明确的组件端口。

在使用 OpenMax IL 时，既可以经由客户端处理数据流，也可以不经由客户端处理数据流。在图 13-7 中，Source 组件到 Host 组件的数据流就是经过客户端的；而 Host 组件到 Accelerator 组件的数据流就没有经过客户端，使用了隧道化的方式；Accelerator 组件和 Sink 组件甚至可以使用私有的通信方式。

OpenMax Core 是辅助组件正常运行的模块，其任务是完成各个组件的初始化等工作。在具体运行时，需要重点初始化 OpenMax IL 组件，而不是初始化 OpenMax Core 组件。

在 OpenMax IL 层中，真正的核心内容是 OpenMAL IL 组件，此组件分别以输入端和输出端为接口，端口可以被连接到另一个组件上。外部对组件可以发送命令，还可以进行设置/获取参数、配置等操作。组件的端口可以包含缓冲区（Buffer）的队列。

在 OpenMax IL 层中，组件处理的核心内容是通过输入端口来消耗 Buffer，通过输出端口来填充 Buffer，这样做的好处是通过多个组件的相互联接构成流式处理。在 OpenMax IL 层中，一个组件的基本结构如图 13-8 所示。



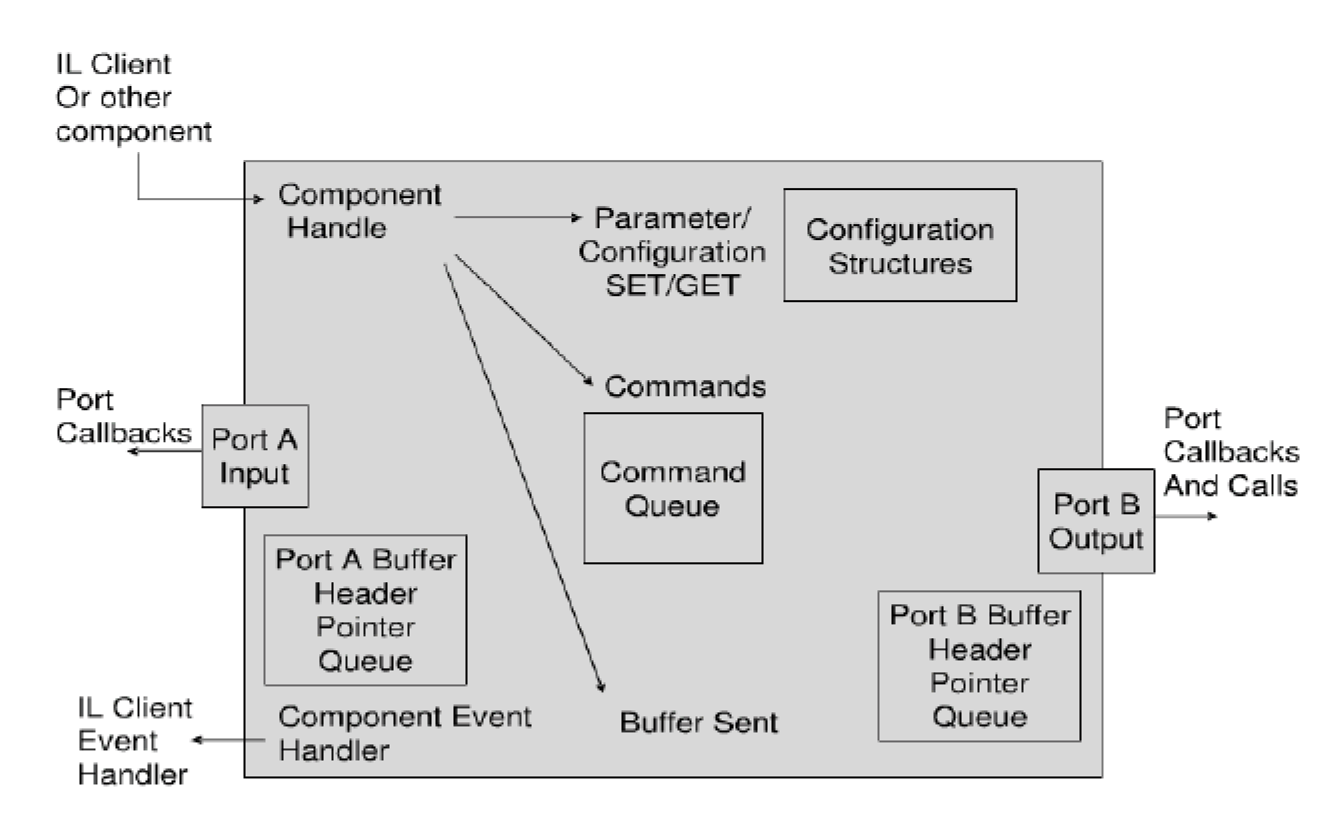


图 13-8　OpenMax IL 层中的组件结构

组件的功能和定义端口的类型有着密切的联系，在大多数情况下的具体联系如下所示。

☑ 只有一个输出端口的是 Source 组件。
☑ 只有一个输入端口的是 Sink 组件。
☑ 有多个输入端口、一个输出端口的是 Mux 组件。
☑ 有一个输入端口、多个输出端口的是 DeMux 组件。输入和输出端口各一个组件的为中间处理环节，这是最常见的组件。

端口根据应用来支持不同的数据类型。假如在输入端和输出端各有一个组件，如果输入端口使用的是 MP3 格式数据，而在输出端口使用的是 PCM 格式数据，那么此组件就是一个 MP3 解码组件。

注意：上述组件连接的方式有一个专业术语——隧道化，通过隧道化（Tunneled）的方式可以将不同组件的一个输入端口和一个输出端口连接到一起，此时会合并两个组件的处理过程并实现共同处理，合而为一。

3．Android 中的 OpenMax

在 Android 系统中，主要使用的是标准 OpenMax IL 层的接口，其中只是进行了简单的封装。通过使用标准的 OpenMax IL 实现，可以很容易地将 OpenMax IL 以插件的形式加入到 Android 系统中。

无论是 OpenCore 引擎，还是 StageFright 引擎，都可以使用 OpenMax 作为多媒体编码/解码器的插件，但是并没有直接使用 OpenMax IL 层提供的纯 C 的接口，而只是对其进行了简单的封装处理。

Android 系统对 OpenMax 支持的力度逐渐增大，在 Android 2.x 版本之后，Android 的框架层开始封装定义 OpenMax IL 层接口，甚至使用 Android 中的 Binder IPC 机制来调用。在 StageFright 中使用了 OpenMax IL 层接口，但是没有使用 OpenCore。OpenCore 使用在 OpenMax IL 层作为编/解码器插件在早期版本中已经使用，Android 框架层封装 OpenMax 接口在后面的版本中才引入。

在 Android 系统中，主要使用了 OpenMax 的编码/解码器功能。在 Android 系统中，使用最多的仍然是编码/解码器组件，尽管 OpenMax 也可以生成输入、输出、文件解析/构建等组件。主要原因有如下两点。

（1）媒体输入/输出环节和系统有很大关系，如果一定要使用 OpenMax 标准则会比较麻烦。

（2）文件解析/构建环节一般不需要使用硬件加速。因为编码/解码器组件最能体现硬件加速环节，所以最常使用。

在 Android 系统中，当实现 OpenMax IL 层和标准的 OpenMax IL 层时需要实现如下两个环节。

☑ 编码/解码器驱动程序：位于 Linux 内核空间，通过 Linux 内核调用驱动程序，调用的驱动程序通常是非标准的驱动程序。
☑ OpenMax IL 层：根据 OpenMax IL 层的标准头文件实现不同功能的组件。

另外，Android 还提供了 OpenMax 的适配层接口，此接口可以对 OpenMax IL 的标准组件进行封装并适配。此接口作为 Android 本地层的接口，可以被 Android 的多媒体引擎随时调用。

13.2.2 实现 OpenMax IL 层接口

在 Android 系统中，主要使用了 OpenMax 的编码/解码器功能，这些功能主要是通过 OpenMax IL 层的接口实现的。本节将详细讲解实现 OpenMax IL 层接口的基本知识。

1. OpenMax IL 层的接口

（1）头文件

在 OpenMax IL 层的接口中定义了若干个头文件，被保存在 frameworks/native/include/media/openmax/目录中。在这些文件中定义了实现 OpenMax IL 层接口的内容，这些头文件的具体说明如下所示。

☑ OMX_Types.h：OpenMax IL 的数据类型定义。
☑ OMX_Core.h：OpenMax IL 核心的 API。
☑ OMX_Component.h：OpenMax IL 组件相关的 API。
☑ OMX_Audio.h：音频相关的常量和数据结构。
☑ OMX_IVCommon.h：图像和视频公共的常量和数据结构。
☑ OMX_Image.h：图像相关的常量和数据结构。
☑ OMX_Video.h：视频相关的常量和数据结构。
☑ OMX_Other.h：其他数据结构（包括 A/V 同步）。
☑ OMX_Index.h：OpenMax IL 定义的数据结构索引。
☑ OMX_ContentPipe.h：内容的管道定义。

在 OpenMax 标准中只有头文件，没有标准的库。

（2）实现过程

在具体实现 OpenMax IL 层的接口时，程序员主要实现包含函数指针的结构体，下面看在上述头文件中的实现流程。

① 在文件 frameworks/native/include/media/openmax/OMX_Component.h 中定义的 OMX_COMPONENTTYPE 结构体是 OpenMax IL 层的核心内容，表示一个组件，其实现代码如下所示。

```
typedef struct OMX_COMPONENTTYPE
{
    OMX_U32 nSize;                          /*定义此结构体的大小*/
    OMX_VERSIONTYPE nVersion;               /*版本号*/
    OMX_PTR pComponentPrivate;              /*此组件的私有数据指针*/
    /*调用者（IL client）设置的指针，用于保存其私有数据，传回给所有的回调函数*/
    OMX_PTR pApplicationPrivate;
    /*下面的函数指针返回 OMX_core.h 中的对应内容*/
    OMX_ERRORTYPE (*GetComponentVersion)(
```



```
            /*获得组件的版本*/
            OMX_IN OMX_HANDLETYPE hComponent,
            OMX_OUT OMX_STRING pComponentName,
            OMX_OUT OMX_VERSIONTYPE* pComponentVersion,
            OMX_OUT OMX_VERSIONTYPE* pSpecVersion,
            OMX_OUT OMX_UUIDTYPE* pComponentUUID);
    OMX_ERRORTYPE (*SendCommand)(                    /*发送命令*/
            OMX_IN   OMX_HANDLETYPE hComponent,
            OMX_IN   OMX_COMMANDTYPE Cmd,
            OMX_IN   OMX_U32 nParam1,
            OMX_IN   OMX_PTR pCmdData);
    OMX_ERRORTYPE (*GetParameter)(                   /*获得参数*/
            OMX_IN   OMX_HANDLETYPE hComponent,
            OMX_IN   OMX_INDEXTYPE nParamIndex,
            OMX_INOUT OMX_PTR pComponentParameterStructure);
    OMX_ERRORTYPE (*SetParameter)(                   /*设置参数*/
            OMX_IN   OMX_HANDLETYPE hComponent,
            OMX_IN   OMX_INDEXTYPE nIndex,
            OMX_IN   OMX_PTR pComponentParameterStructure);
    OMX_ERRORTYPE (*GetConfig)(                      /*获得配置*/
            OMX_IN   OMX_HANDLETYPE hComponent,
            OMX_IN   OMX_INDEXTYPE nIndex,
            OMX_INOUT OMX_PTR pComponentConfigStructure);
    OMX_ERRORTYPE (*SetConfig)(                      /*设置配置*/
            OMX_IN   OMX_HANDLETYPE hComponent,
            OMX_IN   OMX_INDEXTYPE nIndex,
            OMX_IN   OMX_PTR pComponentConfigStructure);
    OMX_ERRORTYPE (*GetExtensionIndex)(              /*转换成 OMX 结构的索引*/
            OMX_IN   OMX_HANDLETYPE hComponent,
            OMX_IN   OMX_STRING cParameterName,
            OMX_OUT OMX_INDEXTYPE* pIndexType);
    OMX_ERRORTYPE (*GetState)(                       /*获得组件当前的状态*/
            OMX_IN   OMX_HANDLETYPE hComponent,
            OMX_OUT OMX_STATETYPE* pState);
    OMX_ERRORTYPE (*ComponentTunnelRequest)(         /*用于连接到另一个组件*/
            OMX_IN   OMX_HANDLETYPE hComp,
            OMX_IN   OMX_U32 nPort,
            OMX_IN   OMX_HANDLETYPE hTunneledComp,
            OMX_IN   OMX_U32 nTunneledPort,
            OMX_INOUT   OMX_TUNNELSETUPTYPE* pTunnelSetup);
    OMX_ERRORTYPE (*UseBuffer)(                      /*为某个端口使用 Buffer*/
            OMX_IN OMX_HANDLETYPE hComponent,
            OMX_INOUT OMX_BUFFERHEADERTYPE** ppBufferHdr,
            OMX_IN OMX_U32 nPortIndex,
            OMX_IN OMX_PTR pAppPrivate,
            OMX_IN OMX_U32 nSizeBytes,
            OMX_IN OMX_U8* pBuffer);
    OMX_ERRORTYPE (*AllocateBuffer)(                 /*在某个端口分配 Buffer*/
            OMX_IN OMX_HANDLETYPE hComponent,
            OMX_INOUT OMX_BUFFERHEADERTYPE** ppBuffer,
            OMX_IN OMX_U32 nPortIndex,
```

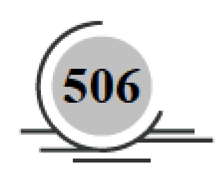
```
                OMX_IN OMX_PTR pAppPrivate,
                OMX_IN OMX_U32 nSizeBytes);
    OMX_ERRORTYPE (*FreeBuffer)(                    /*将某个端口 Buffer 释放*/
                OMX_IN  OMX_HANDLETYPE hComponent,
                OMX_IN  OMX_U32 nPortIndex,
                OMX_IN  OMX_BUFFERHEADERTYPE* pBuffer);
    OMX_ERRORTYPE (*EmptyThisBuffer)(               /*让组件消耗此 Buffer*/
                OMX_IN  OMX_HANDLETYPE hComponent,
                OMX_IN  OMX_BUFFERHEADERTYPE* pBuffer);
    OMX_ERRORTYPE (*FillThisBuffer)(                /*让组件填充此 Buffer*/
                OMX_IN  OMX_HANDLETYPE hComponent,
                OMX_IN  OMX_BUFFERHEADERTYPE* pBuffer);
    OMX_ERRORTYPE (*SetCallbacks)(                  /*设置回调函数 */
                OMX_IN  OMX_HANDLETYPE hComponent,
                OMX_IN  OMX_CALLBACKTYPE* pCallbacks,
                OMX_IN  OMX_PTR pAppData);
    OMX_ERRORTYPE (*ComponentDeInit)(               /*反初始化组件*/
                OMX_IN  OMX_HANDLETYPE hComponent);
    OMX_ERRORTYPE (*UseEGLImage)(
                OMX_IN OMX_HANDLETYPE hComponent,
                OMX_INOUT OMX_BUFFERHEADERTYPE** ppBufferHdr,
                OMX_IN OMX_U32 nPortIndex,
                OMX_IN OMX_PTR pAppPrivate,
                OMX_IN void* eglImage);
    OMX_ERRORTYPE (*ComponentRoleEnum)(
                OMX_IN OMX_HANDLETYPE hComponent,
            OMX_OUT OMX_U8 *cRole,
            OMX_IN OMX_U32 nIndex);
} OMX_COMPONENTTYPE;
```

在实现上述 OMX_COMPONENTTYPE 结构体后，调用者可以使用的内容就是各个函数指针，而这些函数指针和文件 OMX_core.h 中定义的内容相对应。例如，在文件 OMX_core.h 中定义 OMX_FreeBuffer 的代码如下所示。

```
#define OMX_FreeBuffer(
        hComponent,
        nPortIndex,
        pBuffer)
    ((OMX_COMPONENTTYPE*)hComponent)->FreeBuffer(
        hComponent,
        nPortIndex,
        pBuffer)
```

在文件 OMX_core.h 中定义 OMX_FillThisBuffer 的代码如下所示。

```
#define OMX_FillThisBuffer(
        hComponent,
        pBuffer)
    ((OMX_COMPONENTTYPE*)hComponent)->FillThisBuffer(
        hComponent,
        pBuffer)
```



② 接下来需要定义组件运行机制。其中 EmptyThisBuffer 和 FillThisBuffer 是驱动组件运行的基本机制，前者让组件消耗缓冲区，表示对应组件输入的内容；后者让组件填充缓冲区，表示对应组件输出的内容。其中，定义 OMX_EmptyThisBuffer 的代码如下所示。

```
#define OMX_EmptyThisBuffer(
        hComponent,
        pBuffer)
    ((OMX_COMPONENTTYPE*)hComponent)->EmptyThisBuffer(
        hComponent,
        pBuffer)
```

定义 OMX_FillThisBuffer 的代码如下所示。

```
#define OMX_FillThisBuffer(
        hComponent,
        pBuffer)
    ((OMX_COMPONENTTYPE*)hComponent)->FillThisBuffer(
        hComponent,
        pBuffer)
```

③ 开始定义和端口相关的缓冲区管理函数，这些函数分别是 UseBuffer()、AllocateBuffer()和 FreeBuffer()，对于组件的端口有些可以自己分配缓冲区，有些可以使用外部的缓冲区，因此有不同的接口对其进行操作。

④ 使用 SendCommand 向组件发送控制类的命令。接口 GetParameter、SetParameter、GetConfig、SetConfig 用于辅助参数和配置的设置及获取。具体代码如下所示。

```
#define OMX_GetParameter(
        hComponent,
        nParamIndex,
        pComponentParameterStructure)
    ((OMX_COMPONENTTYPE*)hComponent)->GetParameter(
        hComponent,
        nParamIndex,
        pComponentParameterStructure)
#define OMX_SetParameter(
        hComponent,
        nParamIndex,
        pComponentParameterStructure)
    ((OMX_COMPONENTTYPE*)hComponent)->SetParameter(
        hComponent,
        nParamIndex,
        pComponentParameterStructure)
#define OMX_GetConfig(
        hComponent,
        nConfigIndex,
        pComponentConfigStructure)
    ((OMX_COMPONENTTYPE*)hComponent)->GetConfig(
        hComponent,
        nConfigIndex,
        pComponentConfigStructure)
```

```
#define OMX_SetConfig(
        hComponent,
        nConfigIndex,
        pComponentConfigStructure)
    ((OMX_COMPONENTTYPE*)hComponent)->SetConfig(
        hComponent,
        nConfigIndex,
        pComponentConfigStructure)
```

⑤ 使用 ComponentTunnelRequest 实现组件之间的隧道化连接，在此需要指定两个组件及其相连的端口。

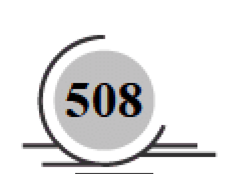
⑥ 接下来使用 ComponentDeInit 反初始化组件。在文件 OMX_Component.h 中定义的端口类型为 OMX_PORTDOMAINTYPE 枚举类型，此枚举的定义代码如下所示。

```
typedef enum OMX_PORTDOMAINTYPE {
    OMX_PortDomainAudio,                    /*音频类型端口*/
    OMX_PortDomainVideo,                    /*视频类型端口*/
    OMX_PortDomainImage,                    /*图像类型端口*/
    OMX_PortDomainOther,                    /*其他类型端口*/
    OMX_PortDomainKhronosExtensions = 0x6F000000,
    OMX_PortDomainVendorStartUnused = 0x7F000000
    OMX_PortDomainMax = 0x7ffffff
} OMX_PORTDOMAINTYPE;
```

在上述代码中，分别定义了音频类型、视频类型和图像类型这 3 种常见类型，至于其他类型则是 OpenMax IL 层所定义的第 4 种端口的类型。

⑦ 使用 OMX_PARAM_PORTDEFINITIONTYPE 类（也在 OMX_Component.h 中定义）定义端口的具体内容，其实现代码如下所示。

```
typedef struct OMX_PARAM_PORTDEFINITIONTYPE {
    OMX_U32 nSize;                          /*结构体大小*/
    OMX_VERSIONTYPE nVersion;               /*版本*/
    OMX_U32 nPortIndex;                     /*端口号*/
    OMX_DIRTYPE eDir;                       /*端口的方向*/
    OMX_U32 nBufferCountActual;             /*为此端口实际分配的 Buffer 的数目*/
    OMX_U32 nBufferCountMin;                /*此端口最小 Buffer 的数目*/
    OMX_U32 nBufferSize;                    /*缓冲区的字节数*/
    OMX_BOOL bEnabled;                      /*是否使能*/
    OMX_BOOL bPopulated;                    /*是否在填充*/
    OMX_PORTDOMAINTYPE eDomain;             /*端口的类型*/
    union {                                 /*端口实际的内容，由类型确定具体结构*/
        OMX_AUDIO_PORTDEFINITIONTYPE audio;
        OMX_VIDEO_PORTDEFINITIONTYPE video;
        OMX_IMAGE_PORTDEFINITIONTYPE image;
        OMX_OTHER_PORTDEFINITIONTYPE other;
    } format;
    OMX_BOOL bBuffersContiguous;
     OMX_U32 nBufferAlignment;
} OMX_PARAM_PORTDEFINITIONTYPE;
```



对于上述代码的具体说明如下所示。

- ☑ OMX_DIRTYPE：端口的方向，包含如下两种。
  - ➢ OMX_DirInput：输入。
  - ➢ OMX_DirOutput：输出。
- ☑ 端口格式的数据结构：使用 format 联合体来表示，具体由如下 4 种不同类型来表示，与端口的类型相对应。
  - ➢ OMX_AUDIO_PORTDEFINITIONTYPE
  - ➢ OMX_VIDEO_PORTDEFINITIONTYPE
  - ➢ OMX_IMAGE_PORTDEFINITIONTYPE
  - ➢ OMX_OTHER_PORTDEFINITIONTYPE

上述类型分别在头文件 OMX_Audio.h、OMX_Video.h、OMX_Image.h 和 OMX_Other.h 中定义。

- ☑ OMX_BUFFERHEADERTYPE：表示一个缓冲区的头部结构，在 OMX_Core.h 中定义。

⑧ 在文件 OMX_Core.h 中定义的枚举类型 OMX_STATETYPE 来表示 OpenMax 的状态，主要代码如下所示。

```
typedef enum OMX_STATETYPE
{
    OMX_StateInvalid,                         /*如果组件监测到内部的数据结构被破坏*/
    OMX_StateLoaded,                          /*如果组件被加载但是没有完成初始化*/
    OMX_StateIdle,                            /*如果组件初始化完成，准备开始*/
    OMX_StateExecuting,                       /*如果组件接受了开始命令，正在创建数据*/
    OMX_StatePause,                           /*如果组件接受暂停命令*/
    OMX_StateWaitForResources,                /*如果组件正在等待资源*/
    OMX_StateKhronosExtensions = 0x6F000000,  /*保留*/
    OMX_StateVendorStartUnused = 0x7F000000,  /*保留*/
    OMX_StateMax = 0X7FFFFFFF
} OMX_STATETYPE;
```

⑨ 在文件 OMX_Core.h 中定义的枚举类型 OMX_COMMANDTYPE，此枚举表示对组件的命令类型，主要代码如下所示。

```
typedef enum OMX_COMMANDTYPE
{
    OMX_CommandStateSet,                          /*改变状态机器*/
    OMX_CommandFlush,                             /*刷新数据队列*/
    OMX_CommandPortDisable,                       /*禁止端口*/
    OMX_CommandPortEnable,                        /*使能端口*/
    OMX_CommandMarkBuffer,                        /*标记组件或 Buffer 用于观察*/
    OMX_CommandKhronosExtensions = 0x6F000000,    /*保留*/
    OMX_CommandVendorStartUnused = 0x7F000000,    /*保留*/
    OMX_CommandMax = 0X7FFFFFFF
} OMX_COMMANDTYPE;
```

注意： 在 OpenMax 的函数参数中，经常包含 OMX_IN 和 OMX_OUT 等宏，其实际内容为空，只是为了标记参数的方向是输入还是输出。

2．在 OpenMax IL 层中工作

在实现 OpenMax IL 层时一般不调用 OpenMax DL 层，具体实现的内容是各个不同的组件。通常通过以下两个步骤来实现 OpenMax IL 组件。

（1）组件的初始化函数

包括硬件和 OpenMax 数据结构的初始化，主要步骤如下所示。

☑ 初始化函数指针。

☑ 初始化私有数据结构。

☑ 初始化端口。

在实现上述步骤的过程中，可以使用其中的 pComponentPrivate 成员保留本组件的私有数据为上下文，在最后获得填充完成 OMX_COMPONENTTYPE 类型的结构体。

（2）OMX_COMPONENTTYPE 类型结构体的各个指针

在此需要实现其中的各个函数指针，当需要用到私有数据时，先从 pComponentPrivate 中得到指针，然后转化成实际的数据结构使用。

因为在 OpenMax IL 层中，经常用到的组件大多数是一个输入端口和一个输出端口，所以端口定义的是 OpenMax IL 组件对外部的接口。对于最常用的编/解码（Codec）组件来说，通常需要在每个组件的实现过程中调用硬件的编/解码接口来实现。在组件的内部处理中可以通过建立线程来处理。在 OpenMax 组件的端口中有默认参数，但也可以在运行时设置，因此一个端口也可以支持不同的编码格式。音频编码组件的输出和输入通常是原始数据格式 PCM，视频编码组件的输出和输入通常是原始数据格式 YUV。

3．OpenMax 适配层

Android 系统中的 OpenMax 适配层的接口在文件 frameworks/av/include/media/IOMX.h 中定义。

文件 IOMX.h 的主要代码如下所示。

```
class IOMX : public IInterface {
public:
    DECLARE_META_INTERFACE(OMX);
    typedef void *buffer_id;
    typedef void *node_id;
    virtual bool livesLocally(pid_t pid) = 0;
    struct ComponentInfo {                                          //组件的信息
        String8 mName;
        List<String8> mRoles;
    };
    virtual status_t listNodes(List<ComponentInfo>    *list) = 0;              //节点列表
    virtual status_t allocateNode(
            const char *name, const sp<IOMXObserver>    &observer,   //分配节点
            node_id *node) = 0;
    virtual status_t freeNode(node_id node) = 0;                    //找到节点
    virtual status_t sendCommand(                                   //发送命令
            node_id node, OMX_COMMANDTYPE cmd, OMX_S32 param) = 0;
    virtual status_t getParameter(                                  //获得参数
            node_id node, OMX_INDEXTYPE index,
            void *params, size_t size) = 0;
```

```
    virtual status_t setParameter(                                    //设置参数
            node_id node, OMX_INDEXTYPE index,
            const void *params, size_t size) = 0;
    virtual status_t getConfig(
            //获得配置
            node_id node, OMX_INDEXTYPE index,
            void *params, size_t size) = 0;
    virtual status_t setConfig(
            //设置配置
            node_id node, OMX_INDEXTYPE index,
            const void *params, size_t size) = 0;
    virtual status_t useBuffer(
            //使用缓冲区
            node_id node, OMX_U32 port_index, const
sp<IMemory> &params,
            buffer_id *buffer) = 0;
    virtual status_t allocateBuffer(
            //分配缓冲区
            node_id node, OMX_U32 port_index, size_t size,
            buffer_id *buffer, void **buffer_data) = 0;
    virtual status_t allocateBufferWithBackup(
            //分配后备缓冲区
            node_id node, OMX_U32 port_index, const
sp<IMemory> &params,
            buffer_id *buffer) = 0;
    virtual status_t freeBuffer(
            //释放缓冲区
            node_id node, OMX_U32 port_index,buffer_id buffer) = 0;
    virtual status_t fillBuffer(node_id node,   buffer_id buffer) = 0;   //填充缓冲区
    virtual status_t emptyBuffer(                                     //消耗缓冲区
            node_id node,
            buffer_id buffer,
            OMX_U32 range_offset, OMX_U32 range_length,
            OMX_U32 flags, OMX_TICKS timestamp) = 0;
    virtual status_t getExtensionIndex(
            node_id node,
            const char *parameter_name,
            OMX_INDEXTYPE *index) = 0;
    virtual sp<IOMXRenderer> createRenderer(                          //创建渲染器（从 ISurface）
            const sp<ISurface> &surface,
            const char *componentName,
            OMX_COLOR_FORMATTYPE colorFormat,
            size_t encodedWidth, size_t encodedHeight,
            size_t displayWidth, size_t displayHeight) = 0;
    sp<IOMXRenderer> createRenderer(                                  //创建渲染器（从 Surface）
            const sp<Surface> &surface,
            const char *componentName,
            OMX_COLOR_FORMATTYPE colorFormat,
            size_t encodedWidth, size_t encodedHeight,
            size_t displayWidth, size_t displayHeight);
    sp<IOMXRenderer> createRendererFromJavaSurface(                   //从 Java 层创建渲染器
```

```
                JNIEnv *env, jobject javaSurface,
                const char *componentName,
                OMX_COLOR_FORMATTYPE colorFormat,
                size_t encodedWidth, size_t encodedHeight,
                size_t displayWidth, size_t displayHeight);
};
```

在 IOMX 中，只有第一个 createRenderer()函数是纯虚函数，第二个函数 createRenderer()和 createRendererFromJavaSurface()通过调用第一个 createRenderer()函数实现。

类 IOMXRenderer 表示了一个 OpenMax 的渲染器，定义此类的代码如下所示。

```
class IOMXRenderer : public IInterface {
public:
    DECLARE_META_INTERFACE(OMXRenderer);
    virtual void render(IOMX::buffer_id buffer) = 0;                //渲染输出函数
};
```

在类 IOMXRenderer 中只包含了一个 Render 接口，其参数类型 IOMX::buffer_id 其实是 void*，可以根据不同的渲染器使用不同的类型。

在文件 IOMX.h 中还存在一个观察器类 IOMXObserver，此类表示 OpenMax 的观察者，其中包含了函数 onMessage()，其参数是 omx_message 接口体，其中包含 Event 事件类型、FillThisBuffer 完成和 EmptyThisBuffer 完成几种类型。

13.3 OpenCore 框架详解

本节将详细讲解 OpenCore 框架的基本知识，分别介绍其结构和插件机制，为读者学习本书后面的知识打下基础。

13.3.1 OpenCore 层次结构

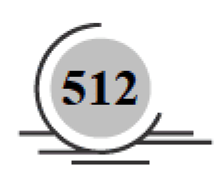
在 Android 系统中，OpenCore 的另外一个名是 PacketVideo，是 Android 多媒体系统的核心。其实 PacketVideo 是一家公司的名称，而 OpenCore 是这套多媒体框架的软件层的名称。在 Android 开发者的眼中，二者的含义基本相同。与其他 Android 程序库相比，OpenCore 的代码非常庞大，是基于 C++ 实现的，定义了全功能的操作系统移植层，各种基本功能均被封装成类的形式，各层次之间的接口使用继承等方式实现。

Android 系统中的 OpenCore 是一个多媒体的框架，从宏观上来看主要包含了如下两方面的内容。

（1）PVPlayer：提供了媒体播放器的功能，可以完成各种音频（Audio）、视频（Video）流的回放（Playback）功能。

（2）PVAuthor：提供了媒体流的记录功能，可以完成各种音频（Audio）、视频（Video）以及静态图像捕获功能。

PVPlayer 和 PVAuthor 以 SDK 的形式提供给开发者，可以在这个 SDK 之上构建多种应用程序和服务。在移动终端中常常使用多媒体应用程序，例如，媒体播放器、照相机、录像机和录音机等。



OpenCore 系统的基本结构如图 13-9 所示。

图 13-9　OpenCore 的基本结构

在图 13-9 所示的结构中，主要层次元素的具体说明如下所示。

（1）OSCL：是 Operating System Compatibility Library 的缩写，意为操作系统兼容库。在 OSCL 中包含了一些操作系统底层的操作，目的是为了更好地在不同操作系统间移植。在 OSCL 中包含的系统底层操作有基本数据类型、配置、字符串工具、I/O、错误处理、线程等，类似一个基础的 C++库。

（2）PVMF：是 Packet Video Multimedia Framework 的缩写，意为 PV 多媒体框架。PVMF 可以在框架内实现一个文件解析（parser）和组成（composer）、编/解码的 NODE，也可以继承其通用的接口，在用户层实现一些 NODE。

（3）PVPlayer Engine：是 PVPlayer 引擎。

（4）PVAuthor Engine：是 PVAuthor 引擎。

除了上述 4 个元素外，OpenCore 中包含的内容还有很多。从播放的角度看，PVPlayer 输入的（Source）是文件或者网络媒体流，输出（Sink）的是音频、视频的输出设备，其基本功能包含了媒体流控制、文件解析、音频/视频流的解码（Decode）等方面的内容。除了从文件中播放媒体文件之外，还包含了与网络相关的 RTSP 流（Real Time Stream Protocol，实时流协议）。在媒体流记录方面，PVAuthor 的输入（Source）是照相机、麦克风等设备，输出（Sink）是各种文件，包含了流的同步、音频/视频流的编码（Encode）以及文件的写入等功能。

在使用 OpenCore SDK 时，有可能需要在应用层实现一个适配器（Adaptor），然后在适配器之上实现具体的功能，对于 PVMF 的 NODE 也可以基于通用的接口在上层实现，以插件的形式使用。

13.3.2　OpenCore 代码结构

在 Android 系统中，OpenCore 的代码保存在 external/opencore/目录中，此目录是 OpenCore 的根目录，其中包含的各个子目录的具体说明如下所示。

（1）android：是一个上层库，基于 PVPlayer 和 PVAuthor 的 SDK 实现了一个为 Android 使用的 Player 和 Author。

（2）baselibs：包含了数据结构和线程安全等内容的底层库。

（3）codecs_v2：是一个内容较多的库，主要包含了编/解码的实现和 OpenMax 的实现。

（4）engines：包含 PVPlayer 和 PVAuthor 引擎的实现。

（5）extern_libs_v2：包含了 khronos 的 OpenMax 的头文件。

（6）fileformats：文件格式的解析（parser）工具。

（7）nodes：提供了 PVMF 的 NODE，主要是编/解码和文件解析方面的 NODE。

（8）oscl：是操作系统兼容库。

（9）pvmi：包含了输入/输出控制的抽象接口。

（10）protocols：主要包含了和网络相关的 RTSP、RTP、HTTP 等协议的内容。

（11）pvcommon：是 pvcommon 库文件的 Android.mk 文件，没有源文件。

（12）pvplayer：是 pvplayer 库文件的 Android.mk 文件，没有源文件。

（13）pvauthor：是 pvauthor 库文件的 Android.mk 文件，没有源文件。

（14）tools_v2：包含了编译工具以及一些可注册的模块。

另外，在 external/opencore/目录中还包含了如下两个文件。

☑ Android.mk：全局的编译文件。

☑ pvplayer.conf：配置文件。

在 external/opencore/的各个子文件夹中还包含了很多 Android.mk 文件，在这些文件之间存在着“递归”的关系。例如，在根目录下的 Android.mk 中包含了下面的内容片断。

☑ include $(PV_TOP)/pvcommon/Android.mk

☑ include $(PV_TOP)/pvplayer/Android.mk

☑ include $(PV_TOP)/pvauthor/Android.mk

这表示要引用 pvcommon、pvplayer 和 pvauthor 等目录下面的 Android.mk 文件。external/opencore/目录中各个 Android.mk 文件可以按照排列组合进行使用，可以将几个 Android.mk 内容合并在一个库中。

13.3.3 OpenCore 编译结构

在 Android 开源系统中，通过 OpenCore 编译的各个库的具体说明如下所示。

☑ libopencoreauthor.so：OpenCore 的 Author 库。

☑ libopencorecommon.so：OpenCore 底层的公共库。

☑ libopencoredownloadreg.so：下载注册库。

☑ libopencoredownload.so：下载功能实现库。

☑ libopencoremp4reg.so：MP4 注册库。

☑ libopencoremp14.so：MP4 功能实现库。

☑ libopencorenet_support.so：网络支持库。

☑ libopencoreplayer.so：OpenCore 的 Player 库。

☑ libopencorertspreg.so：RTSP 注册库。

☑ libopencorertsp.so：RTSP 功能实现库。



OpenCore 中的各个库之间的关系如下所示。

☑ libopencorecommon.so：是所有库的依赖库，提供了公共的功能。

☑ libopencoreplayer.so 和 libopencoreauthor.so：是两个并立的库，分别用于回放和记录，而且这两个库是 OpenCore 对外的接口库。

☑ libopencorenet_support.so：提供网络支持的功能。

除此之外，还有一些功能以插件（Plug-In）的方式放入 Player 中使用，每个功能使用两个库，一个实现具体功能，一个用于注册。下面将简要介绍 OpenCore 中各个库的基本结构。

1. libopencorecommon.so 库的结构

libopencorecommon.so 库是整个 OpenCore 的核心库，其编译控制的文件路径是 pvcommon/Android.mk。

上述文件使用递归的方式寻找子文件，其主要内容如下所示。

```
include $(BUILD_SHARED_LIBRARY)
include $(PV_TOP)//oscl/oscl/osclbase/Android.mk
include $(PV_TOP)//oscl/oscl/osclerror/Android.mk
include $(PV_TOP)//oscl/oscl/osclmemory/Android.mk
include $(PV_TOP)//oscl/oscl/osclutil/Android.mk
include $(PV_TOP)//oscl/pvlogger/Android.mk
include $(PV_TOP)//oscl/oscl/osclproc/Android.mk
include $(PV_TOP)//oscl/oscl/osclio/Android.mk
include $(PV_TOP)//oscl/oscl/osclregcli/Android.mk
include $(PV_TOP)//oscl/oscl/osclregserv/Android.mk
include $(PV_TOP)//oscl/unit_test/Android.mk
include $(PV_TOP)//oscl/oscl/oscllib/Android.mk
include $(PV_TOP)//pvmi/pvmf/Android.mk
include $(PV_TOP)//baselibs/pv_mime_utils/Android.mk
include $(PV_TOP)//nodes/pvfileoutputnode/Android.mk
include $(PV_TOP)//baselibs/media_data_structures/Android.mk
include $(PV_TOP)//baselibs/threadsafe_callback_ao/Android.mk
include $(PV_TOP)//codecs_v2/utilities/colorconvert/Android.mk
include $(PV_TOP)//codecs_v2/audio/gsm_amr/amr_nb/common/Android.mk
include $(PV_TOP)//codecs_v2/video/avc_h264/common/Android.mk
```

这些被包含的 Android.mk 文件真正指定需要编译的文件，这些文件在 Android.mk 的目录及其子目录中。事实上，在 libopencorecommon.so 库中包含了以下内容。

☑ OSCL 的所有内容。

☑ PVMF 框架部分的内容（pvmi/pvmf/Android.mk）。

☑ 基础库中的一些内容（baselibs）。

☑ 编/解码的一些内容。

☑ 文件输出的 node（nodes/pvfileoutputnode /Android.mk）。

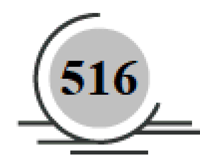
从库 libopencorecommon.so 的结构可以看出，最终生成库的结构与 OpenCore 的层次关系并非完全重合。在库 libopencorecommon.so 中已经包含了底层的 OSCL 的内容、PVMF 的框架以及 Node 和编/解码的工具。

2．libopencoreplayer.so 库的结构

libopencoreplayer.so 库是一个用于实现播放功能的库，其编译控制的文件的路径是 pvplayer/Android.mk。

上述文件 Android.mk 的主要代码如下所示。

```
include $(BUILD_SHARED_LIBRARY)
include $(PV_TOP)//engines/player/Android.mk
include $(PV_TOP)//codecs_v2/audio/aac/dec/util/getactualaacconfig/Android.mk
include $(PV_TOP)//codecs_v2/video/avc_h264/dec/Android.mk
include $(PV_TOP)//codecs_v2/audio/aac/dec/Android.mk
include $(PV_TOP)//codecs_v2/audio/gsm_amr/amr_nb/dec/Android.mk
include $(PV_TOP)//codecs_v2/audio/gsm_amr/amr_wb/dec/Android.mk
include $(PV_TOP)//codecs_v2/audio/gsm_amr/common/dec/Android.mk
include $(PV_TOP)//codecs_v2/audio/mp3/dec/Android.mk
include $(PV_TOP)//codecs_v2/utilities/m4v_config_parser/Android.mk
include $(PV_TOP)//codecs_v2/utilities/pv_video_config_parser/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_common/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_queue/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_h264/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_aac/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_amr/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_mp3/Android.mk
include $(PV_TOP)//codecs_v2/omx/factories/omx_m4v_factory/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_proxy/Android.mk
include $(PV_TOP)//nodes/common/Android.mk
include $(PV_TOP)//pvmi/content_policy_manager/Android.mk
include $(PV_TOP)//pvmi/content_policy_manager/plugins/oma1/passthru/Android.mk
include $(PV_TOP)//pvmi/content_policy_manager/plugins/common/Android.mk
include $(PV_TOP)//pvmi/media_io/pvmiofileoutput/Android.mk
include $(PV_TOP)//fileformats/common/parser/Android.mk
include $(PV_TOP)//fileformats/id3parcom/Android.mk
include $(PV_TOP)//fileformats/rawgsmamr/parser/Android.mk
include $(PV_TOP)//fileformats/mp3/parser/Android.mk
include $(PV_TOP)//fileformats/mp4/parser/Android.mk
include $(PV_TOP)//fileformats/rawaac/parser/Android.mk
include $(PV_TOP)//fileformats/wav/parser/Android.mk
include $(PV_TOP)//nodes/pvaacffparsernode/Android.mk
include $(PV_TOP)//nodes/pvmp3ffparsernode/Android.mk
include $(PV_TOP)//nodes/pvamrffparsernode/Android.mk
include $(PV_TOP)//nodes/pvmediaoutputnode/Android.mk
include $(PV_TOP)//nodes/pvomxvideodecnode/Android.mk
include $(PV_TOP)//nodes/pvomxaudiodecnode/Android.mk
include $(PV_TOP)//nodes/pvwavffparsernode/Android.mk
include $(PV_TOP)//pvmi/recognizer/Android.mk
include $(PV_TOP)//pvmi/recognizer/plugins/pvamrffrecognizer/Android.mk
include $(PV_TOP)//pvmi/recognizer/plugins/pvmp3ffrecognizer/Android.mk
include $(PV_TOP)//pvmi/recognizer/plugins/pvwavffrecognizer/Android.mk
include $(PV_TOP)//engines/common/Android.mk
include $(PV_TOP)//engines/adapters/player/framemetadatautility/Android.mk
```



```
include $(PV_TOP)//protocols/rtp_payload_parser/util/Android.mk
include $(PV_TOP)//android/Android.mk
include $(PV_TOP)//android/drm/oma1/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_rtsp/core/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_rtsp/node_registry/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_net_support/core/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_download/core/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_download/node_registry/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_mp4/core/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_mp4/node_registry/Android.mk
```

在 libopencoreplayer.so 库中包含了如下内容。

☑　解码工具。

☑　文件的解析器（MP4）。

☑　解码工具对应的 Node。

☑　player 的引擎部分（路径是 engines/player/Android.mk）。

☑　为 Android 的 player 适配器（路径是 android/Android.mk）。

☑　识别工具（路径是 pvmi/recognizer）。

☑　编/解码工具中的 OpenMax 部分（路径是 codecs_v2/omx）。

☑　对应几个插件 Node 的注册。

libopencoreplayer.so 库中的内容较多，其中主要为各个文件解析器和解码器，PVPlayer 的核心功能在文件 engines/player/Android.mk 中，而文件 android/Android.mk 的内容比较特殊，是在 PVPlayer 之上构建的一个为 Android 使用的播放器。

3．libopencoreauthor.so 库的结构

libopencoreauthor.so 库是实现媒体流记录的功能库，其编译控制文件的路径是 pvauthor/Android.mk。上述文件 Android.mk 的主要代码如下所示。

```
include $(BUILD_SHARED_LIBRARY)
include $(PV_TOP)//engines/author/Android.mk
include $(PV_TOP)//codecs_v2/video/m4v_h263/enc/Android.mk
include $(PV_TOP)//codecs_v2/audio/gsm_amr/amr_nb/enc/Android.mk
include $(PV_TOP)//codecs_v2/video/avc_h264/enc/Android.mk
include $(PV_TOP)//fileformats/mp4/composer/Android.mk
include $(PV_TOP)//nodes/pvamrencnode/Android.mk
include $(PV_TOP)//nodes/pvmp4ffcomposernode/Android.mk
include $(PV_TOP)//nodes/pvvideoencnode/Android.mk
include $(PV_TOP)//nodes/pvavcencnode/Android.mk
include $(PV_TOP)//nodes/pvmediainputnode/Android.mk
include $(PV_TOP)//android/author/Android.mk
```

在 libopencoreauthor.so 库中包含了如下内容。

☑　编码工具，例如，视频流 H263、H264，音频流 Amr。

☑　文件的组成器，例如 MP4。

☑　编码工具对应的 Node。

☑　用于媒体输入的 Node（目录是 nodes/pvmediainputnode/Android.m）。

☑ Author 引擎（目录是 engines/author/Android.mk）。

☑ Android 的 Author 适配器（目录是 android/author/Android.mk）。

在 libopencoreauthor.so 库中，其内容主要由各个文件编码器和文件组成器构成，其中，PVAuthor 的核心功能保存在 engines/author/Android.mk 目录中，而文件 android/author/Android.mk 是在 PVAuthor 之上构建的一个为 Android 使用的媒体记录器。

4．其他库

除了前面介绍的 3 个库之外，在 OpenCore 中还有另外几个库，具体说明如下所示。

网络支持库 libopencorenet_support.so，对应的 Android.mk 文件的路径是 tools_v2/build/modules/linux_net_support/core/Android.mk。

MP4 功能实现库 libopencoremp14.so 和注册库 libopencoremp4reg.so，对应的 Android.mk 文件的路径是 tools_v2/build/modules/linux_mp4/core/Android.mk 和 tools_v2/build/modules/linux_mp4/node_registry/Android.mk。

RTSP 功能实现库 libopencorertsp.so 和注册库 libopencorertspreg.so，对应的 Android.mk 文件的路径是 tools_v2/build/modules/linux_rtsp/core/Android.mk 和 tools_v2/build/modules/linux_rtsp/node_registry/Android.mk。

下载功能实现库 libopencoredownload.so 和注册库 libopencoredownloadreg.so，对应的 Android.mk 文件的路径是 tools_v2/build/modules/linux_download/core/Android.mk 和 tools_v2/build/modules/linux_download/node_registry/Android.mk。

13.3.4 操作系统兼容库

OSCL（Operating System Compatibility Library，操作系统兼容库）中包含了一些不同操作系统中移植层的功能，其代码结构如下所示。

```
oscl/oscl
|-- config：配置的宏
|-- makefile
|-- makefile.pv
|-- osclbase：包含基本类型、宏以及一些 STL 容器类似的功能
|-- osclerror：错误处理的功能
|-- osclio：文件 I/O 和 Socket 等功能
|-- oscllib：动态库接口等功能
|-- osclmemory：内存管理、自动指针等功能
|-- osclproc：线程、多任务通信等功能
|-- osclregcli：注册客户端的功能
|-- osclregserv：注册服务器的功能
`-- osclutil：字符串等基本功能
```

在目录 oscl 中，通常用一个目录表示一个模块。OSCL 对应的功能非常详细，几乎封装 C 语言中的每一个细节功能，并且提供了 C++接口供上层使用。其实 OperCore 中的 PVMF 和 Engine 都在使用 OSCL，整个 OperCore 的调用者也需要使用 OSCL。

在实现 OSCL 时，简单封装了很多典型的 C 语言函数，例如，osclutil 中与数学相关的功能在



oscl_math.inl 中被定义成了内嵌（inline）的函数，具体代码如下所示。

```
OSCL_COND_EXPORT_REF OSCL_INLINE double oscl_log(double value)
{
		return (double) log(value);
}
OSCL_COND_EXPORT_REF OSCL_INLINE double oscl_log10(double value)
{
		return (double) log10(value);
}
OSCL_COND_EXPORT_REF OSCL_INLINE double oscl_sqrt(double value)
{
		return (double) sqrt(value);
}
```

因为文件 oscl_math.inl 被 oscl_math.h 所包含，所以其结果是和函数 oscl_log()的功能等价的原始函数 log()。

OSCL 的具体实现比较复杂，很多 C 语言标准库的句柄都被定义成了 C++类的形式，实现起来会比较繁琐。尽管如此，OSCL 的复杂性不是很高。以 oscllib 为例，其代码结构如下所示。

```
oscl/oscl/oscllib/
|-- Android.mk
|-- build
|    `-- make
|         `-- makefile
`-- src
     |-- oscl_library_common.h
     |-- oscl_library_list.cpp
     |-- oscl_library_list.h
     |-- oscl_shared_lib_interface.h
     |-- oscl_shared_library.cpp
     `-- oscl_shared_library.h
```

其中，文件 oscl_shared_library.h 是提供给上层使用的动态库的接口功能，定义的接口代码如下所示。

```
class OsclSharedLibrary {
public:
		OSCL_IMPORT_REF OsclSharedLibrary();
		OSCL_IMPORT_REF OsclSharedLibrary(const OSCL_String& aPath);
		OSCL_IMPORT_REF ~OsclSharedLibrary();
		OSCL_IMPORT_REF OsclLibStatus LoadLib(const OSCL_String& aPath);
		OSCL_IMPORT_REF OsclLibStatus LoadLib();
		OSCL_IMPORT_REF void SetLibPath(const OSCL_String& aPath);
		OSCL_IMPORT_REF OsclLibStatus QueryInterface(const OsclUuid& aInterfaceId, OsclAny*&
aInterfacePtr);
		OSCL_IMPORT_REF OsclLibStatus Close();
		OSCL_IMPORT_REF void AddRef();
		OSCL_IMPORT_REF void RemoveRef();
}
```

这些接口都与库的加载有关系，而在文件 oscl_shared_library.cpp 中，其具体的功能通过使用函数

dlopen()等来实现。

13.3.5　实现 OpenCore 中的 OpenMax 部分

在 OpenCore 框架中，OpenMax 是作为插件来实现的，只要封装了 OpenMax，就可以在 OpenCore 中使用标准的 OpenMax。

1．OpenMax 结构

在 OpenCore 中，在目录 extern_libs_v2/khronos/openmax/include/的头文件中包含标准的 OpenMax。

在文件 build_config/opencore_dynamic/Android_omx_aacdec_sharedlibrary.mk 中声明了插件 OpenMax 的主要库是 libomx_sharedlibrary.so，主要代码如下所示。

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_WHOLE_STATIC_LIBRARIES := \
	libomx_aac_component_lib \
	libpv_aac_dec
LOCAL_MODULE := libomx_aacdec_sharedlibrary
-include $(PV_TOP)/Android_platform_extras.mk
-include $(PV_TOP)/Android_system_extras.mk

LOCAL_SHARED_LIBRARIES += libomx_sharedlibrary libopencore_common

include $(BUILD_SHARED_LIBRARY)
include $(PV_TOP)/codecs_v2/omx/omx_aac/Android.mk
include $(PV_TOP)/codecs_v2/audio/aac/dec/Android.mk
```

库 libomx_sharedlibrary.so 为 omx 针对 OpenCore 的接口层库，也就是说在每个模拟器上 libomx_sharedlibrary.so 向外（即 OpenCore）提供的接口是一致的。此库可以动态打开各个 OpenMax 的编/解码模块，各个编/解码模块通过调用 codecs_v2 中 audio 和 video 目录中软件的编/解码库来实现。

在 opencore 的根目录中，有一个名为 pvplayer.cfg 的文件，此文件用于实现 OpenCore 运行过程的动态配置，此文件的主要代码如下所示。

```
(0x1d4769f0,0xca0c,0x11dc,0x95,0xff,0x08,0x00,0x20,0x0c,0x9a,0x66),"libopencore_rtspreg.so"
(0x1d4769f0,0xca0c,0x11dc,0x95,0xff,0x08,0x00,0x20,0x0c,0x9a,0x66),"libopencore_downloadreg.so"
(0x1d4769f0,0xca0c,0x11dc,0x95,0xff,0x08,0x00,0x20,0x0c,0x9a,0x66),"libopencore_mp4localreg.so"
(0x6d3413a0,0xca0c,0x11dc,0x95,0xff,0x08,0x00,0x20,0x0c,0x9a,0x66),"libopencore_mp4localreg.so"
(0xa054369c,0x22c5,0x412e,0x19,0x17,0x87,0x4c,0x1a,0x19,0xd4,0x5f),"libomx_sharedlibrary.so"
```

2．OpenMax 接口

在 OpenCore 中，OpenMax 接口是通过封装标准的 OpenMax IL 层来构建的，这些接口的基本内容相同，但是不同于标准的 OpenMax IL 层的 C 语言接口。在 OpenCore 中和 OpenMax 接口相关的头文件如下所示。

- ☑ opencore/codecs_v2/omx/omx_mastercore/include/omx_interface.h：定义插件接口。
- ☑ opencore/codecs_v2/omx/omx_common/include/pv_omxcore.h：核心定义。

☑　opencore/codecs_v2/omx/omx_baseclass/include/pv_omxcomponent.h：定义 PV 的 OpenMax 组件。

文件 omx_interface.h 定义了 OpenMax 接口的核心功能，在其中包含了各种函数指针的定义类型，具体实现代码如下所示。

```
typedef OMX_ERRORTYPE OMX_APIENTRY(*tpOMX_Init)(void);
typedef OMX_ERRORTYPE OMX_APIENTRY(*tpOMX_Deinit)(void);
typedef OMX_ERRORTYPE OMX_APIENTRY(*tpOMX_ComponentNameEnum)(
    OMX_OUT OMX_STRING cComponentName,
    OMX_IN OMX_U32 nNameLength,
    OMX_IN OMX_U32 nIndex);
typedef OMX_ERRORTYPE OMX_APIENTRY(*tpOMX_GetHandle)(
    OMX_OUT OMX_HANDLETYPE* pHandle,
    OMX_IN OMX_STRING cComponentName,
    OMX_IN OMX_PTR pAppData,
    OMX_IN OMX_CALLBACKTYPE* pCallBacks);
typedef OMX_ERRORTYPE OMX_APIENTRY(*tpOMX_FreeHandle)(
    OMX_IN OMX_HANDLETYPE hComponent);
typedef OMX_ERRORTYPE(*tpOMX_GetComponentsOfRole)(
    OMX_IN OMX_STRING role,
    OMX_INOUT OMX_U32 *pNumComps,
    OMX_INOUT OMX_U8    **compNames);
typedef OMX_ERRORTYPE(*tpOMX_GetRolesOfComponent)(
    OMX_IN OMX_STRING compName,
    OMX_INOUT OMX_U32 *pNumRoles,
    OMX_OUT OMX_U8 **roles);
typedef OMX_ERRORTYPE OMX_APIENTRY(*tpOMX_SetupTunnel)(
    OMX_IN OMX_HANDLETYPE hOutput,
    OMX_IN OMX_U32 nPortOutput,
    OMX_IN OMX_HANDLETYPE hInput,
    OMX_IN OMX_U32 nPortInput);
typedef OMX_ERRORTYPE(*tpOMX_GetContentPipe)(
    OMX_OUT OMX_HANDLETYPE *hPipe,
    OMX_IN OMX_STRING szURI);

typedef OMX_BOOL(*tpOMXConfigParser)(
    OMX_PTR aInputParameters,
    OMX_PTR aOutputParameters);
```

上述函数指针是 OpenMax 的核心方法，这些指针类型需要使用继承来设置。

另外，在文件 omx_interface.h 中还定义了类 OMXInterface，在类中包含了一系列函数，这些函数返回的都是上面类型的函数指针。类 OMXInterface 是 OpenMax 直接实现 OpenCore 的接口。

```
class OMXInterface : public OsclSharedLibraryInterface
{
    public:
        OMXInterface()
        {
            pOMX_Init = NULL;
            pOMX_Deinit = NULL;
            pOMX_ComponentNameEnum = NULL;
```

```
        pOMX_GetHandle = NULL;
        pOMX_FreeHandle = NULL;
        pOMX_GetComponentsOfRole = NULL;
        pOMX_GetRolesOfComponent = NULL;
        pOMX_SetupTunnel = NULL;
        pOMX_GetContentPipe = NULL;
        pOMXConfigParser = NULL;
    };
    virtual bool UnloadWhenNotUsed(void) = 0;
    tpOMX_Init GetpOMX_Init()
    {
        return pOMX_Init;
    };
    tpOMX_Deinit GetpOMX_Deinit()
    {
        return   pOMX_Deinit;
    };
    tpOMX_ComponentNameEnum GetpOMX_ComponentNameEnum()
    {
        return pOMX_ComponentNameEnum;
    };
    tpOMX_GetHandle GetpOMX_GetHandle()
    {
        return pOMX_GetHandle;
    };
    tpOMX_FreeHandle GetpOMX_FreeHandle()
    {
        return pOMX_FreeHandle;
    };
    tpOMX_GetComponentsOfRole GetpOMX_GetComponentsOfRole()
    {
        return pOMX_GetComponentsOfRole;
    };
    tpOMX_GetRolesOfComponent GetpOMX_GetRolesOfComponent()
    {
        return pOMX_GetRolesOfComponent;
    };
    tpOMX_SetupTunnel GetpOMX_SetupTunnel()
    {
        return pOMX_SetupTunnel;
    };
    tpOMX_GetContentPipe GetpOMX_GetContentPipe()
    {
        return pOMX_GetContentPipe;
    };
    tpOMXConfigParser GetpOMXConfigParser()
    {
        return pOMXConfigParser;
    };
    OMX_ERRORTYPE OMX_APIENTRY(*pOMX_Init)(void);
    OMX_ERRORTYPE OMX_APIENTRY(*pOMX_Deinit)(void);
```

```
        OMX_ERRORTYPE OMX_APIENTRY(*pOMX_ComponentNameEnum)(
            OMX_OUT OMX_STRING cComponentName,
            OMX_IN OMX_U32 nNameLength,
            OMX_IN OMX_U32 nIndex);
        OMX_ERRORTYPE OMX_APIENTRY(*pOMX_GetHandle)(
            OMX_OUT OMX_HANDLETYPE* pHandle,
            OMX_IN OMX_STRING cComponentName,
            OMX_IN OMX_PTR pAppData,
            OMX_IN OMX_CALLBACKTYPE* pCallBacks);
        OMX_ERRORTYPE OMX_APIENTRY(*pOMX_FreeHandle)(
            OMX_IN OMX_HANDLETYPE hComponent);
        OMX_ERRORTYPE(*pOMX_GetComponentsOfRole)(
            OMX_IN OMX_STRING role,
            OMX_INOUT OMX_U32 *pNumComps,
            OMX_INOUT OMX_U8    **compNames);
        OMX_ERRORTYPE(*pOMX_GetRolesOfComponent)(
            OMX_IN OMX_STRING compName,
            OMX_INOUT OMX_U32 *pNumRoles,
            OMX_OUT OMX_U8 **roles);
        OMX_ERRORTYPE OMX_APIENTRY(*pOMX_SetupTunnel)(
            OMX_IN OMX_HANDLETYPE hOutput,
            OMX_IN OMX_U32 nPortOutput,
            OMX_IN OMX_HANDLETYPE hInput,
            OMX_IN OMX_U32 nPortInput);
        OMX_ERRORTYPE(*pOMX_GetContentPipe)(
            OMX_OUT OMX_HANDLETYPE *hPipe,
            OMX_IN OMX_STRING szURI);
        OMX_BOOL(*pOMXConfigParser)(
            OMX_PTR aInputParameters,
            OMX_PTR aOutputParameters);
};
```

3．OpenMax 组织结构

在文件 opencore/codecs_v2/omx/omx_sharedlibrary/interface/src/pv_omx_interface.cpp 中实现了类 OMXInterface，在实现时是通过实现类中的函数指针的方式实现的。

在文件 pv_omx_interface.cpp 中，函数 PVGetInterface()和 PVReleaseInterface()是使用 C 语言导出的函数，这两个函数的实现代码如下所示。

```
extern "C"
{
    OSCL_EXPORT_REF OsclAny* PVGetInterface()
    {
        return PVOMXInterface::Instance();
    }
    OSCL_EXPORT_REF void PVReleaseInterface(void* interface)
    {
        PVOMXInterface* pInterface = (PVOMXInterface*)interface;
        if (pInterface)
        {
```

```
                OSCL_DELETE(pInterface);
            }
        }
}
```

在文件 pv_omx_interface.cpp 中，类 PVOMXInterface 继承了 OMXInterface，在此类的构造函数中设置了各个 OMXInterface 中的函数指针。构造函数 PVOMXInterface()的主要代码如下所示。

```
private:
        PVOMXInterface()
        {
            //设置指针 OMX 的核心方法
            pOMX_Init = OMX_Init;
            pOMX_Deinit = OMX_Deinit;
            pOMX_ComponentNameEnum = OMX_ComponentNameEnum;
            pOMX_GetHandle = OMX_GetHandle;
            pOMX_FreeHandle = OMX_FreeHandle;
            pOMX_GetComponentsOfRole = OMX_GetComponentsOfRole;
            pOMX_GetRolesOfComponent = OMX_GetRolesOfComponent;
            pOMX_SetupTunnel = OMX_SetupTunnel;
            pOMX_GetContentPipe = OMX_GetContentPipe;
            pOMXConfigParser = OMXConfigParser;
        };
```

上述构造函数都是在文件 opencore/codecs_v2/omx/omx_common/src/pv_omxcore.cpp 中实现的，此文件实现了 OpenMax 的核心功能。

文件 opencore/codecs_v2/omx/omx_common/src/pv_omxregistry.cpp 的功能是注册 OpenMax 模块，其主要实现代码如下所示。

```
//注册 MP3 解码器
OMX_ERRORTYPE Mp3Register()
{
    ComponentRegistrationType  *pCRT  =  (ComponentRegistrationType  *)  oscl_malloc(sizeof(Component
RegistrationType));
    if (pCRT)
    {
        pCRT->ComponentName = (OMX_STRING)"OMX.PV.mp3dec";//组件名
        pCRT->RoleString[0] = (OMX_STRING)"audio_decoder.mp3";
        pCRT->NumberOfRolesSupported = 1;
        pCRT->SharedLibraryOsclUuid = NULL;
#if USE_DYNAMIC_LOAD_OMX_COMPONENTS
        pCRT->FunctionPtrCreateComponent = &OmxComponentFactoryDynamicCreate;
        pCRT->FunctionPtrDestroyComponent = &OmxComponentFactoryDynamicDestructor;
        pCRT->SharedLibraryName = (OMX_STRING)"libomx_mp3dec_sharedlibrary.so";
        pCRT->SharedLibraryPtr = NULL;
        OsclUuid *temp = (OsclUuid *) oscl_malloc(sizeof(OsclUuid));
        if (temp == NULL)
        {
            oscl_free(pCRT); //释放内存
            return OMX_ErrorInsufficientResources;
```

```
        }
        OSCL_PLACEMENT_NEW(temp, PV_OMX_MP3DEC_UUID);
        pCRT->SharedLibraryOsclUuid = (OMX_PTR) temp;
        pCRT->SharedLibraryRefCounter = 0;
#endif
#if REGISTER_OMX_MP3_COMPONENT
#if (DYNAMIC_LOAD_OMX_MP3_COMPONENT == 0)
        pCRT->FunctionPtrCreateComponent = &Mp3OmxComponentFactory;
        pCRT->FunctionPtrDestroyComponent = &Mp3OmxComponentDestructor;
        pCRT->SharedLibraryName = NULL;
        pCRT->SharedLibraryPtr = NULL;
        if (pCRT->SharedLibraryOsclUuid)
            oscl_free(pCRT->SharedLibraryOsclUuid);
        pCRT->SharedLibraryOsclUuid = NULL;
        pCRT->SharedLibraryRefCounter = 0;
#endif
#endif
    }
    else
    {
        return OMX_ErrorInsufficientResources;
    }
    return   ComponentRegister(pCRT);
}
//WMA 格式解码
OMX_ERRORTYPE WmaRegister()
{
    ComponentRegistrationType *pCRT = (ComponentRegistrationType *) oscl_malloc(sizeof(Component
RegistrationType));
    if (pCRT)
    {
        pCRT->ComponentName = (OMX_STRING)"OMX.PV.wmadec";
        pCRT->RoleString[0] = (OMX_STRING)"audio_decoder.wma";
        pCRT->NumberOfRolesSupported = 1;
        pCRT->SharedLibraryOsclUuid = NULL;
#if USE_DYNAMIC_LOAD_OMX_COMPONENTS
        pCRT->FunctionPtrCreateComponent = &OmxComponentFactoryDynamicCreate;
        pCRT->FunctionPtrDestroyComponent = &OmxComponentFactoryDynamicDestructor;
        pCRT->SharedLibraryName = (OMX_STRING)"libomx_wmadec_sharedlibrary.so";
        pCRT->SharedLibraryPtr = NULL;
        OsclUuid *temp = (OsclUuid *) oscl_malloc(sizeof(OsclUuid));
        if (temp == NULL)
        {
            oscl_free(pCRT); // free allocated memory
            return OMX_ErrorInsufficientResources;
        }
        OSCL_PLACEMENT_NEW(temp, PV_OMX_WMADEC_UUID);
        pCRT->SharedLibraryOsclUuid = (OMX_PTR) temp;
        pCRT->SharedLibraryRefCounter = 0;
#endif
#if REGISTER_OMX_WMA_COMPONENT
```

```
#if (DYNAMIC_LOAD_OMX_WMA_COMPONENT == 0)

            pCRT->FunctionPtrCreateComponent = &WmaOmxComponentFactory;
            pCRT->FunctionPtrDestroyComponent = &WmaOmxComponentDestructor;
            pCRT->SharedLibraryName = NULL;
            pCRT->SharedLibraryPtr = NULL;
            if (pCRT->SharedLibraryOsclUuid)
                  oscl_free(pCRT->SharedLibraryOsclUuid);
            pCRT->SharedLibraryOsclUuid = NULL;
            pCRT->SharedLibraryRefCounter = 0;
#endif
#endif
      }
      else
      {
            return OMX_ErrorInsufficientResources;
      }
      return    ComponentRegister(pCRT);
}
```

4．实现 OpenMax 编/解码组件

OpenMax 的主要功能是通过解/编码组件实现的，各个组件的基本结构类似，其实现内容实际上就是文件 opencore/codecs_v2/omx/omx_baseclass/include/pv_omxcomponent.h 中定义的类 OmxComponentBase。假如要实现 MP3 格式文件的解码处理，则在目录 opencore/codecs_v2/omx/mp3 中实现 MP3 的解码功能。

在上述目录中，文件 Android.mk 生成了名为 libomx_mp3_component_lib.so 的库，此静态库将被连接生成动态库 libomx_mp3dec_sharedlibrary_lib。此 Android.mk 文件的主要代码如下所示。

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_SRC_FILES := \
       src/mp3_dec.cpp \
       src/omx_mp3_component.cpp \
       src/mp3_timestamp.cpp
LOCAL_MODULE := libomx_mp3_component_lib
LOCAL_CFLAGS :=    $(PV_CFLAGS)
LOCAL_ARM_MODE := arm
LOCAL_STATIC_LIBRARIES :=
LOCAL_SHARED_LIBRARIES :=
LOCAL_C_INCLUDES := \
       $(PV_TOP)/codecs_v2/omx/omx_mp3/src \
       $(PV_TOP)/codecs_v2/omx/omx_mp3/include \
       $(PV_TOP)/extern_libs_v2/khronos/openmax/include \
       $(PV_TOP)/codecs_v2/omx/omx_baseclass/include \
       $(PV_TOP)/codecs_v2/audio/mp3/dec/src \
       $(PV_TOP)/codecs_v2/audio/mp3/dec/include \
       $(PV_INCLUDES)
LOCAL_COPY_HEADERS_TO := $(PV_COPY_HEADERS_TO)
LOCAL_COPY_HEADERS := \
       include/mp3_dec.h \
```

```
	include/omx_mp3_component.h \
	include/mp3_timestamp.h
include $(BUILD_STATIC_LIBRARY)
```

在目录 opencore/codecs_v2/omx/omx_mp3/src/中存在了如下 3 个文件。

☑ mp3_dec.cpp：能够调用 MP3 解码器组件。

☑ mp3_timestamp.cpp：能够实现时间戳功能。

☑ omx_mp3_component.cpp：定义了 MP3 解码器组件。

在文件 opencore/codecs_v2/omx/omx_mp3/include/omx_mp3_component.h 中定义了类 OpenmaxMp3AO，此类继承了 OmxComponentAudio，主要代码如下所示。

```
class OpenmaxMp3AO : public OmxComponentAudio {
	public:
		OpenmaxMp3AO();
		~OpenmaxMp3AO();
		OMX_ERRORTYPE ConstructComponent(OMX_PTR pAppData, OMX_PTR pProxy);
		OMX_ERRORTYPE DestroyComponent();
		OMX_ERRORTYPE ComponentInit();
		OMX_ERRORTYPE ComponentDeInit();
		static void ComponentGetRolesOfComponent(OMX_STRING* aRoleString);
		void ProcessData();
		void SyncWithInputTimestamp();
		void ProcessInBufferFlag();
		void ResetComponent();
		OMX_ERRORTYPE GetConfig(
			OMX_IN	OMX_HANDLETYPE hComponent,
			OMX_IN	OMX_INDEXTYPE nIndex,
			OMX_INOUT OMX_PTR pComponentConfigStructure);
	private:
		void CheckForSilenceInsertion();
		void DoSilenceInsertion();
		Mp3Decoder* ipMp3Dec;
		Mp3TimeStampCalc iCurrentFrameTS;
};
```

在文件 omx_mp3_component.cpp 中定义了 MP3 解码器组件，通过函数 ProcessData()实现 MP3 文件的解码处理。函数 ProcessData()的实现代码如下所示。

```
void OpenmaxMp3AO::ProcessData()
{
	PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE, (0, "OpenmaxMp3AO :
ProcessData IN"));

	QueueType* pInputQueue	= ipPorts[OMX_PORT_INPUTPORT_INDEX]->pBufferQueue;
	QueueType* pOutputQueue = ipPorts[OMX_PORT_OUTPUTPORT_INDEX]->pBufferQueue;

	ComponentPortType* pInPort = (ComponentPortType*) ipPorts[OMX_PORT_INPUTPORT_INDEX];
	ComponentPortType* pOutPort = ipPorts[OMX_PORT_OUTPUTPORT_INDEX];
	OMX_COMPONENTTYPE* pHandle = &iOmxComponent;
```

```
    OMX_U8* pOutBuffer;//输出缓冲区的指针
    OMX_U32 OutputLength; //输出缓冲区的长度
    OMX_S32 DecodeReturn;
    OMX_BOOL ResizeNeeded = OMX_FALSE;

    OMX_U32 TempInputBufferSize = (2 * sizeof(uint8) * (ipPorts[OMX_PORT_INPUTPORT_INDEX]->
PortParam.nBufferSize));

    if ((!iIsInputBufferEnded) || iEndofStream)
    {
        if (OMX_TRUE == iSilenceInsertionInProgress)
        {
            DoSilenceInsertion();
            if (OMX_TRUE == iSilenceInsertionInProgress)
            {
                return;
            }
        }
        //证实 prev 是否发布了 buffer
        if (OMX_TRUE == iNewOutBufRequired)
        {
            //证实是否一个新的输出缓冲区是可利用的
            if (0 == (GetQueueNumElem(pOutputQueue)))
            {
                PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE, (0,
"OpenmaxMp3AO : ProcessData OUT output buffer unavailable"));
                return;
            }
            ipOutputBuffer = (OMX_BUFFERHEADERTYPE*) DeQueue(pOutputQueue);
            if (NULL == ipOutputBuffer)
            {
                PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE, (0,
"OpenmaxMp3AO : ProcessData Error, Output Buffer Dequeue returned NULL, OUT"));
                return;
            }
            ipOutputBuffer->nFilledLen = 0;
            iNewOutBufRequired = OMX_FALSE;
            //设置当前时间戳对输出缓冲区时间戳
            ipOutputBuffer->nTimeStamp = iCurrentFrameTS.GetConvertedTs();
            //复制在动态重组之前当地被存放的输出缓冲区
            //被接受的新的 OMX 缓冲
            if (OMX_TRUE == iSendOutBufferAfterPortReconfigFlag)
            {
                if ((ipTempOutBufferForPortReconfig)
                        && (iSizeOutBufferForPortReconfig <= ipOutputBuffer->nAllocLen))
                {
                    oscl_memcpy(ipOutputBuffer->pBuffer, ipTempOutBufferForPortReconfig, iSizeOutBuffer
ForPortReconfig);
                    ipOutputBuffer->nFilledLen = iSizeOutBufferForPortReconfig;
                    ipOutputBuffer->nTimeStamp = iTimestampOutBufferForPortReconfig;
                }
```

```
                    iSendOutBufferAfterPortReconfigFlag = OMX_FALSE;
                    //只有当充满时退还输出缓冲区
                    if ((ipOutputBuffer->nAllocLen - ipOutputBuffer->nFilledLen) < iOutputFrameLength)
                    {
                        ReturnOutputBuffer(ipOutputBuffer, pOutPort);
                    }
                    //释放临时输出缓冲区
                    if (ipTempOutBufferForPortReconfig)
                    {
                        oscl_free(ipTempOutBufferForPortReconfig);
                        ipTempOutBufferForPortReconfig = NULL;
                        iSizeOutBufferForPortReconfig = 0;
                    }
                    if (OMX_TRUE == iNewOutBufRequired)
                    {
                        if (0 == (GetQueueNumElem(pOutputQueue)))
                        {
                            PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE,
(0, "OpenmaxMp3AO : ProcessData OUT, output buffer unavailable"));
                            return;
                        }
                        ipOutputBuffer = (OMX_BUFFERHEADERTYPE*) DeQueue(pOutputQueue);
                        if (NULL == ipOutputBuffer)
                        {
                            PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE,
(0, "OpenmaxMp3AO : ProcessData Error, Output Buffer Dequeue returned NULL, OUT"));
                            return;
                        }
                        ipOutputBuffer->nFilledLen = 0;
                        iNewOutBufRequired = OMX_FALSE;
                        ipOutputBuffer->nTimeStamp = iCurrentFrameTS.GetConvertedTs();
                    }
                }
            }
            /*标号缓冲的代码
             * 根据 hMarkTargetComponent 设置规格
             */
            if (ipMark != NULL)
            {
                ipOutputBuffer->hMarkTargetComponent = ipMark->hMarkTargetComponent;
                ipOutputBuffer->pMarkData = ipMark->pMarkData;
                ipMark = NULL;
            }
            if (ipTargetComponent != NULL)
            {
                ipOutputBuffer->hMarkTargetComponent = ipTargetComponent;
                ipOutputBuffer->pMarkData = iTargetMarkData;
                ipTargetComponent = NULL;
            }
            //在此标记缓冲代码末端
            pOutBuffer = &ipOutputBuffer->pBuffer[ipOutputBuffer->nFilledLen];
```

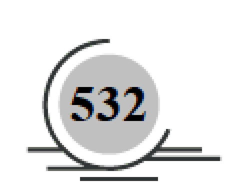
```
            OutputLength = 0;
            /*复制从在临时被存放的前一个输入缓冲区的残余数据
             *缓冲接踵而来的数据流
             */
            if (iTempInputBufferLength > 0 &&
                    ((iInputCurrLength + iTempInputBufferLength) < TempInputBufferSize))
            {
                oscl_memcpy(&ipTempInputBuffer[iTempInputBufferLength], ipFrameDecodeBuffer, iInputCurrLength);
                iInputCurrLength += iTempInputBufferLength;
                iTempInputBufferLength = 0;
                ipFrameDecodeBuffer = ipTempInputBuffer;
            }
            //将输出缓冲区作为指针
            DecodeReturn = ipMp3Dec->Mp3DecodeAudio(//设置 ipMp3Dec 的类型是 Mp3Decode
(OMX_S16*) pOutBuffer,//输出缓冲区的指针
 (OMX_U32*) & OutputLength,//输出缓冲区的长度
                                                    &(ipFrameDecodeBuffer),
                                                    &iInputCurrLength,
                                                    &iFrameCount,
                                                    &(ipPorts[OMX_PORT_OUTPUTPORT_INDEX]->Audio
PcmMode),
                                                    &(ipPorts[OMX_PORT_INPUTPORT_INDEX]->Audio
Mp3Param),
                                                    iEndOfFrameFlag,
                                                    &ResizeNeeded);
            if (ResizeNeeded == OMX_TRUE)
            {
                if (0 != OutputLength)
                {
                    iOutputFrameLength = OutputLength * 2;
                    //更新时间戳
                    iSamplesPerFrame = OutputLength / ipPorts[OMX_PORT_OUTPUTPORT_INDEX]->Audio
PcmMode.nChannels;
                    iCurrentFrameTS.SetParameters(ipPorts[OMX_PORT_OUTPUTPORT_INDEX]->AudioPcmMode.
nSamplingRate, iSamplesPerFrame);
                    iOutputMilliSecPerFrame = iCurrentFrameTS.GetFrameDuration();
                }
                iResizePending = OMX_TRUE;
                /*不要退回引起的输出缓冲区，当地存放并且等待动态接口重新完成构造*/
                if ((NULL == ipTempOutBufferForPortReconfig))
                {
                    ipTempOutBufferForPortReconfig = (OMX_U8*) oscl_malloc(sizeof(uint8) * OutputLength * 2);
                    if (NULL == ipTempOutBufferForPortReconfig)
                    {
                        PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE, (0,
"OpenmaxMp3AO : ProcessData error, insufficient resources"));
                        return;
                    }
                }
                //复制 omx 输出缓冲区对临时内部缓冲
                oscl_memcpy(ipTempOutBufferForPortReconfig, pOutBuffer, OutputLength * 2);
```

```
            iSizeOutBufferForPortReconfig = OutputLength * 2;
            //设置当前时间戳对第一个产品框架的输出缓冲区时间戳
            //以后将取消
            iTimestampOutBufferForPortReconfig = iCurrentFrameTS.GetConvertedTs();
            iCurrentFrameTS.UpdateTimestamp(iSamplesPerFrame);
                OutputLength = 0;
            OMX_COMPONENTTYPE* pHandle = (OMX_COMPONENTTYPE*) ipAppPriv->CompHandle;
            (*(ipCallbacks->EventHandler))
            (pHandle,
             iCallbackData,
             OMX_EventPortSettingsChanged, //The command was completed
             OMX_PORT_OUTPUTPORT_INDEX,
             0,
             NULL);
        }
        ipOutputBuffer->nFilledLen += OutputLength * 2;
        ipOutputBuffer->nOffset = 0;
        if (OutputLength > 0)
        {
            iCurrentFrameTS.UpdateTimestamp(iSamplesPerFrame);
        }
        if (OMX_TRUE == iEndofStream)
        {
         if (MP3DEC_SUCCESS != DecodeReturn)
            {
                PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE, (0,
"OpenmaxMp3AO : ProcessData EOS callback send"));
                (*(ipCallbacks->EventHandler))
                (pHandle,
                 iCallbackData,
                 OMX_EventBufferFlag,
                 1,
                 OMX_BUFFERFLAG_EOS,
                 NULL);
                iEndofStream = OMX_FALSE;
                ipOutputBuffer->nFlags |= OMX_BUFFERFLAG_EOS;
                ReturnOutputBuffer(ipOutputBuffer, pOutPort);
                ipOutputBuffer = NULL;
                PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE, (0,
"OpenmaxMp3AO : ProcessData OUT"));
                return;
            }
        }
        if (MP3DEC_SUCCESS == DecodeReturn)
        {
            ipInputBuffer->nFilledLen = iInputCurrLength;
        }
        else if (MP3DEC_INCOMPLETE_FRAME == DecodeReturn)
        {
            oscl_memcpy(ipTempInputBuffer, ipFrameDecodeBuffer, iInputCurrLength);
            iTempInputBufferLength = iInputCurrLength;
```

```
                ipInputBuffer->nFilledLen = 0;
                iInputCurrLength = 0;
            }
            else
            {
                ipInputBuffer->nFilledLen = 0;
                iInputCurrLength = 0;
                PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE, (0,
"OpenmaxMp3AO : ProcessData ErrorStreamCorrupt callback send"));
                (*(ipCallbacks->EventHandler))
                (pHandle,
                 iCallbackData,
                 OMX_EventError,
                 OMX_ErrorStreamCorrupt,
                 0,
                 NULL);
            }
            //如果经过译码器处理，则会得到充分的消耗并退回到输入缓冲区
            if (0 == ipInputBuffer->nFilledLen)
            {
                ReturnInputBuffer(ipInputBuffer, pInPort);
                ipInputBuffer = NULL;
                iIsInputBufferEnded = OMX_TRUE;
                iInputCurrLength = 0;
            }
            //当充满时送回输出缓冲区
            if ((ipOutputBuffer->nAllocLen - ipOutputBuffer->nFilledLen) < (iOutputFrameLength))
            {
                ReturnOutputBuffer(ipOutputBuffer, pOutPort);
                ipOutputBuffer = NULL;
            }
            //如果有些处理在当前缓冲中，则重新编排 AO
            if (((iInputCurrLength != 0 || GetQueueNumElem(pInputQueue) > 0)
                    && (GetQueueNumElem(pOutputQueue) > 0) && (ResizeNeeded == OMX_FALSE))
                    || (OMX_TRUE == iEndofStream))
            {
                RunIfNotReady();
            }
        }
        PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE, (0, "OpenmaxMp3AO :
ProcessData OUT"));
        return;
}
```

13.4 StageFright 框架详解

在 Android 系统中，预设的多媒体框架（Multimedia Framework）是 OpenCore。OpenCore 的特点是兼顾了跨平台的移植性，而且已经过多方验证，所以相对来说比较稳定；但是其缺点是庞大复杂，



需要耗费相当多的时间去维护。从 Android 2.0 开始，Google 引入了架构稍微简单的 StageFright，并且有逐渐取代 OpenCore 的趋势。从 Android 2.2 开始，几乎完全放弃了 OpenCore，而主推 StageFright。本节将详细讲解 StageFright 框架的基本知识，为读者学习本书后面的知识打下基础。

13.4.1　StageFright 代码结构

StageFright 是一个轻量级的多媒体框架，其主要功能是基于 OpenMax 实现的。在 StageFright 中提供了媒体播放等接口，这些接口可以为 Android 框架层所使用。

在 Android 开源代码中，StageFright 的头文件路径为 frameworks/av/include/media/stagefright/。

实现 StageFright 功能的文件路径为 frameworks/av/media/libstagefright/。

实现 StageFright 播放器和录音器功能的文件路径为 frameworks/av/media/libmediaplayerservice/。

测试 StageFright 功能的代码路径为 frameworks/av/cmds/stagefright/。

13.4.2　StageFright 实现 OpenMax 接口

在 Android 系统中，StageFright 可以实现 OpenMax 的接口，可以让 StageFright 引擎内的 OMXCode 调用实现的 OpenMax 接口，最终目的是使用 OpenMax IL 实现编/解码功能。

在 Android 系统中通过 StageFright 来定义 OpenMax 接口，具体实现内容保存在 omx 目录中。在头文件 frameworks/av/media/libstagefright/include/OMX.h 中实现了 Android 标准的 IOMX 类，此文件的主要代码如下所示。

```
class OMX : public BnOMX,
            public IBinder::DeathRecipient {
public:
    OMX();
    virtual bool livesLocally(pid_t pid);
    virtual status_t listNodes(List<ComponentInfo> *list);
    virtual status_t allocateNode(
            const char *name, const sp<IOMXObserver> &observer, node_id *node);
    virtual status_t freeNode(node_id node);
    virtual status_t sendCommand(
            node_id node, OMX_COMMANDTYPE cmd, OMX_S32 param);
    virtual status_t getParameter(
            node_id node, OMX_INDEXTYPE index,
            void *params, size_t size);
…
    virtual status_t emptyBuffer(
            node_id node,
            buffer_id buffer,
            OMX_U32 range_offset, OMX_U32 range_length,
            OMX_U32 flags, OMX_TICKS timestamp);
    virtual status_t getExtensionIndex(
            node_id node,
            const char *parameter_name,
            OMX_INDEXTYPE *index);
    virtual sp<IOMXRenderer> createRenderer(
```

```
            const sp<ISurface> &surface,
            const char *componentName,
            OMX_COLOR_FORMATTYPE colorFormat,
            size_t encodedWidth, size_t encodedHeight,
            size_t displayWidth, size_t displayHeight,
            int32_t rotationDegrees);
```

文件 frameworks/av/media/libstagefright/omx/OMX.cpp 是上述 OMX.h 文件的实现文件，首先定义函数 createRenderer()来创建映射，先建立一个 hardware renderer—SharedVideoRenderer(libstagefrighthw.so)，如果失败，则建立 software renderer—SoftwareRenderer(surface)。此函数的主要代码如下所示。

```
sp<IOMXRenderer> OMX::createRenderer(
        const sp<ISurface> &surface,
        const char *componentName,
        OMX_COLOR_FORMATTYPE colorFormat,
        size_t encodedWidth, size_t encodedHeight,
        size_t displayWidth, size_t displayHeight,
        int32_t rotationDegrees) {
    Mutex::Autolock autoLock(mLock);
    VideoRenderer *impl = NULL;
    void *libHandle = dlopen("libstagefrighthw.so", RTLD_NOW);
    if (libHandle) {
        typedef VideoRenderer *(*CreateRendererWithRotationFunc)(
                const sp<ISurface> &surface,
                const char *componentName,
                OMX_COLOR_FORMATTYPE colorFormat,
                size_t displayWidth, size_t displayHeight,
                size_t decodedWidth, size_t decodedHeight,
                int32_t rotationDegrees);
        typedef VideoRenderer *(*CreateRendererFunc)(
                const sp<ISurface> &surface,
                const char *componentName,
                OMX_COLOR_FORMATTYPE colorFormat,
                size_t displayWidth, size_t displayHeight,
                size_t decodedWidth, size_t decodedHeight);
        CreateRendererWithRotationFunc funcWithRotation =
            (CreateRendererWithRotationFunc)dlsym(
                    libHandle,
                    "_Z26createRendererWithRotationRKN7android2spINS_8"
                    "ISurfaceEEEPKc20OMX_COLOR_FORMATTYPEjjjji");
        if (funcWithRotation) {
            impl = (*funcWithRotation)(
                    surface, componentName, colorFormat,
                    displayWidth, displayHeight, encodedWidth, encodedHeight,
                    rotationDegrees);
        } else {
            CreateRendererFunc func =
                (CreateRendererFunc)dlsym(
                        libHandle,
                        "_Z14createRendererRKN7android2spINS_8ISurfaceEEEPKc20"
                        "OMX_COLOR_FORMATTYPEjjjj");
```

```
            if (func) {
                impl = (*func)(surface, componentName, colorFormat,
                        displayWidth, displayHeight, encodedWidth, encodedHeight);
            }
        }
        if (impl) {
            impl = new SharedVideoRenderer(libHandle, impl);
            libHandle = NULL;
        }
        if (libHandle) {
            dlclose(libHandle);
            libHandle = NULL;
        }
    }
    if (!impl) {
        LOGW("Using software renderer.");
        impl = new SoftwareRenderer(
                colorFormat,
                surface,
                displayWidth, displayHeight,
                encodedWidth, encodedHeight);
        if (((SoftwareRenderer *)impl)->initCheck() != OK) {
            delete impl;
            impl = NULL;
            return NULL;
        }
    }
    return new OMXRenderer(impl);
}
```

由此可见，OMXMaster 是 OMX.cpp 的真正实现者，并且能够管理 OpenMax 插件的类，这些功能是通过头文件 OMXMaster.h 和源码文件 OMXMaster.cpp 实现的。其中在文件 frameworks/av/include/media/stagefright/OMXMaster.h 中定义了类 OMXMaster，主要代码如下所示。

```
struct OMXCodec : public MediaSource,
                  public MediaBufferObserver {
    enum CreationFlags {
        kPreferSoftwareCodecs = 1,
        kIgnoreCodecSpecificData = 2,
        kClientNeedsFramebuffer = 4,
    };
    static sp<MediaSource> Create(          //创建类 MediaSource
            const sp<IOMX> &omx,
            const sp<MetaData> &meta, bool createEncoder,
            const sp<MediaSource> &source,
            const char *matchComponentName = NULL,
            uint32_t flags = 0);
    static void setComponentRole(           //设置组件的职责
            const sp<IOMX> &omx, IOMX::node_id node, bool isEncoder,
            const char *mime);
    virtual status_t start(MetaData *params = NULL);
```

```
    virtual status_t stop();
    virtual sp<MetaData> getFormat();
//省略声明函数代码
...
```

在文件 frameworks/av/media/libstagefright/OMXMaster.cpp 中定义静态函数 Create()将 MediaSource 作为 IOMX 插件给 OMXCode。函数 Create()的主要实现代码如下所示。

```
sp<MediaSource> OMXCodec::Create(
        const sp<IOMX> &omx,
        const sp<MetaData> &meta, bool createEncoder,
        const sp<MediaSource> &source,
        const char *matchComponentName,
        uint32_t flags) {
    const char *mime;
    bool success = meta->findCString(kKeyMIMEType, &mime);//获取 mime 信息
    CHECK(success);
    Vector<String8> matchingCodecs;
    findMatchingCodecs(
            mime, createEncoder, matchComponentName, flags, &matchingCodecs);
    if (matchingCodecs.isEmpty()) {
        return NULL;
    }
    sp<OMXCodecObserver> observer = new OMXCodecObserver;
    IOMX::node_id node = 0;
    const char *componentName;
    for (size_t i = 0; i < matchingCodecs.size(); ++i) {            //使用 for 循环查找插件
        componentName = matchingCodecs[i].string();
        sp<MediaSource> softwareCodec = createEncoder?
            InstantiateSoftwareEncoder(componentName, source, meta):
            InstantiateSoftwareCodec(componentName, source);
        if (softwareCodec != NULL) {
            LOGV("Successfully allocated software codec '%s'", componentName);
            return softwareCodec;
        }
        LOGV("Attempting to allocate OMX node '%s'", componentName);
        uint32_t quirks = getComponentQuirks(componentName, createEncoder);
        if (!createEncoder
                && (quirks & kOutputBuffersAreUnreadable)
                && (flags & kClientNeedsFramebuffer)) {
            if (strncmp(componentName, "OMX.SEC.", 8)) {
                LOGW("Component '%s' does not give the client access to "
                     "the framebuffer contents. Skipping.",
                     componentName);
                continue;
            }
        }
        status_t err = omx->allocateNode(componentName, observer, &node);
        if (err == OK) {
            LOGV("Successfully allocated OMX node '%s'", componentName);
            sp<OMXCodec> codec = new OMXCodec(            //新建类 OMXCodec
```

```
                    omx, node, quirks,
                    createEncoder, mime, componentName,
                    source);
            observer->setCodec(codec);                        //设置编/解码器
            err = codec->configureCodec(meta, flags);
            if (err == OK) {
                return codec;
            }
            LOGV("Failed to configure codec '%s'", componentName);
        }
    }
    return NULL;
}
```

13.4.3　分析 Video Buffer 传输流程

视频播放的过程是处理 Video Buffer 的过程，在 StageFright 框架中需要使用 VideoRenderer 插件来实现处理功能。下面将详细讲解在 StageFright 框架中使用插件来传输 Video Buffer 的具体流程。

（1）文件 OMXCodec.cpp 会在一开始时通过函数 read()来传送未解码的 data 数据给 decoder，并要求 decoder 将解码后的 data 传回来。对应的实现代码如下所示。

```
status_t OMXCodec::read(
        MediaBuffer **buffer, const ReadOptions *options) {
    status_t err = OK;
    *buffer = NULL;

    Mutex::Autolock autoLock(mLock);

    if (mState != EXECUTING && mState != RECONFIGURING) {
        return UNKNOWN_ERROR;
    }

    bool seeking = false;
    int64_t seekTimeUs;
    ReadOptions::SeekMode seekMode;
    if (options && options->getSeekTo(&seekTimeUs, &seekMode)) {
        seeking = true;
    }

    if (mInitialBufferSubmit) {
        mInitialBufferSubmit = false;

        if (seeking) {
            CHECK(seekTimeUs >= 0);
            mSeekTimeUs = seekTimeUs;
            mSeekMode = seekMode;

            seeking = false;
            mPaused = false;
```

```
        }

        drainInputBuffers();

        if (mState == EXECUTING) {
            fillOutputBuffers();
        }
    }

    if (seeking) {
        while (mState == RECONFIGURING) {
            if ((err = waitForBufferFilled_l()) != OK) {
                return err;
            }
        }

        if (mState != EXECUTING) {
            return UNKNOWN_ERROR;
        }

        CODEC_LOGV("seeking to %lld us (%.2f secs)", seekTimeUs, seekTimeUs / 1E6);

        mSignalledEOS = false;

        CHECK(seekTimeUs >= 0);
        mSeekTimeUs = seekTimeUs;
        mSeekMode = seekMode;

        mFilledBuffers.clear();

        CHECK_EQ((int)mState, (int)EXECUTING);

        bool emulateInputFlushCompletion = !flushPortAsync(kPortIndexInput);
        bool emulateOutputFlushCompletion = !flushPortAsync(kPortIndexOutput);

        if (emulateInputFlushCompletion) {
            onCmdComplete(OMX_CommandFlush, kPortIndexInput);
        }

        if (emulateOutputFlushCompletion) {
            onCmdComplete(OMX_CommandFlush, kPortIndexOutput);
        }

        while (mSeekTimeUs >= 0) {
            if ((err = waitForBufferFilled_l()) != OK) {
                return err;
            }
        }
    }

    while (mState != ERROR && !mNoMoreOutputData && mFilledBuffers.empty()) {
```

```
            if ((err = waitForBufferFilled_l()) != OK) {
                return err;
            }
        }

        if (mState == ERROR) {
            return UNKNOWN_ERROR;
        }

        if (mFilledBuffers.empty()) {
            return mSignalledEOS ? mFinalStatus : ERROR_END_OF_STREAM;
        }

        if (mOutputPortSettingsHaveChanged) {
            mOutputPortSettingsHaveChanged = false;

            return INFO_FORMAT_CHANGED;
        }

        size_t index = *mFilledBuffers.begin();
        mFilledBuffers.erase(mFilledBuffers.begin());

        BufferInfo *info = &mPortBuffers[kPortIndexOutput].editItemAt(index);
        CHECK_EQ((int)info->mStatus, (int)OWNED_BY_US);
        info->mStatus = OWNED_BY_CLIENT;

        info->mMediaBuffer->add_ref();
        if (mSkipCutBuffer != NULL) {
            mSkipCutBuffer->submit(info->mMediaBuffer);
        }
        *buffer = info->mMediaBuffer;

        return OK;
}
void OMXCodec::drainInputBuffers() {
        CHECK(mState == EXECUTING || mState == RECONFIGURING);

        if (mFlags & kUseSecureInputBuffers) {
            Vector<BufferInfo> *buffers = &mPortBuffers[kPortIndexInput];
            for (size_t i = 0; i < buffers->size(); ++i) {
                if (!drainAnyInputBuffer()
                        || (mFlags & kOnlySubmitOneInputBufferAtOneTime)) {
                    break;
                }
            }
        } else {
            Vector<BufferInfo> *buffers = &mPortBuffers[kPortIndexInput];
            for (size_t i = 0; i < buffers->size(); ++i) {
                BufferInfo *info = &buffers->editItemAt(i);

                if (info->mStatus != OWNED_BY_US) {
```

```
                continue;
            }

            if (!drainInputBuffer(info)) {
                break;
            }

            if (mFlags & kOnlySubmitOneInputBufferAtOneTime) {
                break;
            }
        }
    }
}
void OMXCodec::drainInputBuffer(BufferInfo *info)
{
  mOMX->emptyBuffer(...);
}
void OMXCodec::fillOutputBuffers() {
    CHECK_EQ((int)mState, (int)EXECUTING);
    if (mSignalledEOS
            && countBuffersWeOwn(mPortBuffers[kPortIndexInput])
                == mPortBuffers[kPortIndexInput].size()
            && countBuffersWeOwn(mPortBuffers[kPortIndexOutput])
                == mPortBuffers[kPortIndexOutput].size()) {
        mNoMoreOutputData = true;
        mBufferFilled.signal();

        return;
    }

    Vector<BufferInfo> *buffers = &mPortBuffers[kPortIndexOutput];
    for (size_t i = 0; i < buffers->size(); ++i) {
        BufferInfo *info = &buffers->editItemAt(i);
        if (info->mStatus == OWNED_BY_US) {
            fillOutputBuffer(&buffers->editItemAt(i));
        }
    }
}
void OMXCodec::fillOutputBuffer(BufferInfo *info) {
    CHECK_EQ((int)info->mStatus, (int)OWNED_BY_US);

    if (mNoMoreOutputData) {
        CODEC_LOGV("There is no more output data available, not "
             "calling fillOutputBuffer");
        return;
    }

    CODEC_LOGV("Calling fillBuffer on buffer %p", info->mBuffer);
    status_t err = mOMX->fillBuffer(mNode, info->mBuffer);

    if (err != OK) {
```

```
        CODEC_LOGE("fillBuffer failed w/ error 0x%08x", err);

        setState(ERROR);
        return;
    }

    info->mStatus = OWNED_BY_COMPONENT;
}
```

（2）Decoder 从 input port（输入端）获取资料，然后进行解码处理，并回传 EmptyBufferDone 以通知 OMXCodec 当前的工作。对应的实现代码如下所示。

```
void OMXCodec::on_message(const omx_message &msg) {
    if (mState == ERROR) {
        /*
         * only drop EVENT messages, EBD and FBD are still
         * processed for bookkeeping purposes
         */
        if (msg.type == omx_message::EVENT) {
            ALOGW("Dropping OMX EVENT message - we're in ERROR state.");
            return;
        }
    }

    switch (msg.type) {
        case omx_message::EVENT:
        {
            onEvent(
                 msg.u.event_data.event, msg.u.event_data.data1,
                 msg.u.event_data.data2);

            break;
        }

        case omx_message::EMPTY_BUFFER_DONE:
        {
            IOMX::buffer_id buffer = msg.u.extended_buffer_data.buffer;

            CODEC_LOGV("EMPTY_BUFFER_DONE(buffer: %p)", buffer);

            Vector<BufferInfo> *buffers = &mPortBuffers[kPortIndexInput];
            size_t i = 0;
            while (i < buffers->size() && (*buffers)[i].mBuffer != buffer) {
                ++i;
            }

            CHECK(i < buffers->size());
            if ((*buffers)[i].mStatus != OWNED_BY_COMPONENT) {
                ALOGW("We already own input buffer %p, yet received "
                     "an EMPTY_BUFFER_DONE.", buffer);
            }
```

```
            BufferInfo* info = &buffers->editItemAt(i);
            info->mStatus = OWNED_BY_US;

            if (info->mMediaBuffer != NULL) {
                info->mMediaBuffer->release();
                info->mMediaBuffer = NULL;
            }

            if (mPortStatus[kPortIndexInput] == DISABLING) {
                CODEC_LOGV("Port is disabled, freeing buffer %p", buffer);

                status_t err = freeBuffer(kPortIndexInput, i);
                CHECK_EQ(err, (status_t)OK);
            } else if (mState != ERROR
                    && mPortStatus[kPortIndexInput] != SHUTTING_DOWN) {
                CHECK_EQ((int)mPortStatus[kPortIndexInput], (int)ENABLED);

                if (mFlags & kUseSecureInputBuffers) {
                    drainAnyInputBuffer();
                } else {
                    drainInputBuffer(&buffers->editItemAt(i));
                }
            }
            break;
        }

        case omx_message::FILL_BUFFER_DONE:
        {
            IOMX::buffer_id buffer = msg.u.extended_buffer_data.buffer;
            OMX_U32 flags = msg.u.extended_buffer_data.flags;

            CODEC_LOGV("FILL_BUFFER_DONE(buffer: %p, size: %ld, flags: 0x%08lx, timestamp: %lld us
(%.2f secs))",
                 buffer,
                 msg.u.extended_buffer_data.range_length,
                 flags,
                 msg.u.extended_buffer_data.timestamp,
                 msg.u.extended_buffer_data.timestamp / 1E6);

            Vector<BufferInfo> *buffers = &mPortBuffers[kPortIndexOutput];
            size_t i = 0;
            while (i < buffers->size() && (*buffers)[i].mBuffer != buffer) {
                ++i;
            }

            CHECK(i < buffers->size());
            BufferInfo *info = &buffers->editItemAt(i);

            if (info->mStatus != OWNED_BY_COMPONENT) {
                ALOGW("We already own output buffer %p, yet received "
```

```
                "a FILL_BUFFER_DONE.", buffer);
        }

        info->mStatus = OWNED_BY_US;

        if (mPortStatus[kPortIndexOutput] == DISABLING) {
            CODEC_LOGV("Port is disabled, freeing buffer %p", buffer);

            status_t err = freeBuffer(kPortIndexOutput, i);
            CHECK_EQ(err, (status_t)OK);

#if 0
        } else if (mPortStatus[kPortIndexOutput] == ENABLED
                   && (flags & OMX_BUFFERFLAG_EOS)) {
            CODEC_LOGV("No more output data.");
            mNoMoreOutputData = true;
            mBufferFilled.signal();
#endif
        } else if (mPortStatus[kPortIndexOutput] != SHUTTING_DOWN) {
            CHECK_EQ((int)mPortStatus[kPortIndexOutput], (int)ENABLED);

            if (info->mMediaBuffer == NULL) {
                CHECK(mOMXLivesLocally);
                CHECK(mQuirks & kRequiresAllocateBufferOnOutputPorts);
                CHECK(mQuirks & kDefersOutputBufferAllocation);

                info->mMediaBuffer = new MediaBuffer(
                        msg.u.extended_buffer_data.data_ptr,
                        info->mSize);
                info->mMediaBuffer->setObserver(this);
            }

            MediaBuffer *buffer = info->mMediaBuffer;
            bool isGraphicBuffer = buffer->graphicBuffer() != NULL;

            if (!isGraphicBuffer
                && msg.u.extended_buffer_data.range_offset
                    + msg.u.extended_buffer_data.range_length
                        > buffer->size()) {
                CODEC_LOGE(
                        "Codec lied about its buffer size requirements, "
                        "sending a buffer larger than the originally "
                        "advertised size in FILL_BUFFER_DONE!");
            }
            buffer->set_range(
                    msg.u.extended_buffer_data.range_offset,
                    msg.u.extended_buffer_data.range_length);

            buffer->meta_data()->clear();

            buffer->meta_data()->setInt64(
```

```
                kKeyTime, msg.u.extended_buffer_data.timestamp);

        if (msg.u.extended_buffer_data.flags & OMX_BUFFERFLAG_SYNCFRAME) {
            buffer->meta_data()->setInt32(kKeyIsSyncFrame, true);
        }
        bool isCodecSpecific = false;
        if (msg.u.extended_buffer_data.flags & OMX_BUFFERFLAG_CODECCONFIG) {
            buffer->meta_data()->setInt32(kKeyIsCodecConfig, true);
            isCodecSpecific = true;
        }

        if (isGraphicBuffer || mQuirks & kOutputBuffersAreUnreadable) {
            buffer->meta_data()->setInt32(kKeyIsUnreadable, true);
        }

        buffer->meta_data()->setPointer(
                kKeyPlatformPrivate,
                msg.u.extended_buffer_data.platform_private);

        buffer->meta_data()->setPointer(
                kKeyBufferID,
                msg.u.extended_buffer_data.buffer);

        if (msg.u.extended_buffer_data.flags & OMX_BUFFERFLAG_EOS) {
            CODEC_LOGV("No more output data.");
            mNoMoreOutputData = true;
        }

        if (mIsEncoder && mIsVideo) {
            int64_t decodingTimeUs = isCodecSpecific? 0: getDecodingTimeUs();
            buffer->meta_data()->setInt64(kKeyDecodingTime, decodingTimeUs);
        }

        if (mTargetTimeUs >= 0) {
            CHECK(msg.u.extended_buffer_data.timestamp <= mTargetTimeUs);

            if (msg.u.extended_buffer_data.timestamp < mTargetTimeUs) {
                CODEC_LOGV(
                        "skipping output buffer at timestamp %lld us",
                        msg.u.extended_buffer_data.timestamp);

                fillOutputBuffer(info);
                break;
            }

            CODEC_LOGV(
                    "returning output buffer at target timestamp "
                    "%lld us",
                    msg.u.extended_buffer_data.timestamp);

            mTargetTimeUs = -1;
```

```
                }

                mFilledBuffers.push_back(i);
                mBufferFilled.signal();
                if (mIsEncoder) {
                    sched_yield();
                }
            }

            break;
        }

        default:
        {
            CHECK(!"should not be here.");
            break;
        }
    }
}
```

（3）当 OMXCodec 接收到 EMPTY_BUFFER_DONE 之后，继续传送下一个未解码的资料给 Decoder。Decoder 解码后的资料送到 output port（输出端），并回传 FillBufferDone 以通知 OMXCodec。对应的实现代码如下所示。

```
void OMXCodec::on_message(const omx_message &msg) {
    if (mState == ERROR) {
        /*
         * only drop EVENT messages, EBD and FBD are still
         * processed for bookkeeping purposes
         */
        if (msg.type == omx_message::EVENT) {
            ALOGW("Dropping OMX EVENT message - we're in ERROR state.");
            return;
        }
    }

    switch (msg.type) {
        case omx_message::EVENT:
        {
            onEvent(
                 msg.u.event_data.event, msg.u.event_data.data1,
                 msg.u.event_data.data2);

            break;
        }

        case omx_message::EMPTY_BUFFER_DONE:
        {
            IOMX::buffer_id buffer = msg.u.extended_buffer_data.buffer;

            CODEC_LOGV("EMPTY_BUFFER_DONE(buffer: %p)", buffer);
```

```
        Vector<BufferInfo> *buffers = &mPortBuffers[kPortIndexInput];
        size_t i = 0;
        while (i < buffers->size() && (*buffers)[i].mBuffer != buffer) {
            ++i;
        }

        CHECK(i < buffers->size());
        if ((*buffers)[i].mStatus != OWNED_BY_COMPONENT) {
            ALOGW("We already own input buffer %p, yet received "
                 "an EMPTY_BUFFER_DONE.", buffer);
        }

        BufferInfo* info = &buffers->editItemAt(i);
        info->mStatus = OWNED_BY_US;

        if (info->mMediaBuffer != NULL) {
            info->mMediaBuffer->release();
            info->mMediaBuffer = NULL;
        }

        if (mPortStatus[kPortIndexInput] == DISABLING) {
            CODEC_LOGV("Port is disabled, freeing buffer %p", buffer);

            status_t err = freeBuffer(kPortIndexInput, i);
            CHECK_EQ(err, (status_t)OK);
        } else if (mState != ERROR
                && mPortStatus[kPortIndexInput] != SHUTTING_DOWN) {
            CHECK_EQ((int)mPortStatus[kPortIndexInput], (int)ENABLED);

            if (mFlags & kUseSecureInputBuffers) {
                drainAnyInputBuffer();
            } else {
                drainInputBuffer(&buffers->editItemAt(i));
            }
        }
        break;
    }

    case omx_message::FILL_BUFFER_DONE:
    {
        IOMX::buffer_id buffer = msg.u.extended_buffer_data.buffer;
        OMX_U32 flags = msg.u.extended_buffer_data.flags;

        CODEC_LOGV("FILL_BUFFER_DONE(buffer: %p, size: %ld, flags: 0x%08lx, timestamp: %lld us
(%.2f secs))",
             buffer,
             msg.u.extended_buffer_data.range_length,
             flags,
             msg.u.extended_buffer_data.timestamp,
             msg.u.extended_buffer_data.timestamp / 1E6);
```

```
        Vector<BufferInfo> *buffers = &mPortBuffers[kPortIndexOutput];
        size_t i = 0;
        while (i < buffers->size() && (*buffers)[i].mBuffer != buffer) {
            ++i;
        }

        CHECK(i < buffers->size());
        BufferInfo *info = &buffers->editItemAt(i);

        if (info->mStatus != OWNED_BY_COMPONENT) {
            ALOGW("We already own output buffer %p, yet received "
                 "a FILL_BUFFER_DONE.", buffer);
        }

        info->mStatus = OWNED_BY_US;

        if (mPortStatus[kPortIndexOutput] == DISABLING) {
            CODEC_LOGV("Port is disabled, freeing buffer %p", buffer);

            status_t err = freeBuffer(kPortIndexOutput, i);
            CHECK_EQ(err, (status_t)OK);

#if 0
        } else if (mPortStatus[kPortIndexOutput] == ENABLED
                   && (flags & OMX_BUFFERFLAG_EOS)) {
            CODEC_LOGV("No more output data.");
            mNoMoreOutputData = true;
            mBufferFilled.signal();
#endif
        } else if (mPortStatus[kPortIndexOutput] != SHUTTING_DOWN) {
            CHECK_EQ((int)mPortStatus[kPortIndexOutput], (int)ENABLED);

            if (info->mMediaBuffer == NULL) {
                CHECK(mOMXLivesLocally);
                CHECK(mQuirks & kRequiresAllocateBufferOnOutputPorts);
                CHECK(mQuirks & kDefersOutputBufferAllocation);

                info->mMediaBuffer = new MediaBuffer(
                        msg.u.extended_buffer_data.data_ptr,
                        info->mSize);
                info->mMediaBuffer->setObserver(this);
            }

            MediaBuffer *buffer = info->mMediaBuffer;
            bool isGraphicBuffer = buffer->graphicBuffer() != NULL;

            if (!isGraphicBuffer
                && msg.u.extended_buffer_data.range_offset
                    + msg.u.extended_buffer_data.range_length
                        > buffer->size()) {
```

```
        CODEC_LOGE(
                "Codec lied about its buffer size requirements, "
                "sending a buffer larger than the originally "
                "advertised size in FILL_BUFFER_DONE!");
    }
    buffer->set_range(
            msg.u.extended_buffer_data.range_offset,
            msg.u.extended_buffer_data.range_length);

    buffer->meta_data()->clear();

    buffer->meta_data()->setInt64(
            kKeyTime, msg.u.extended_buffer_data.timestamp);

    if (msg.u.extended_buffer_data.flags & OMX_BUFFERFLAG_SYNCFRAME) {
        buffer->meta_data()->setInt32(kKeyIsSyncFrame, true);
    }
    bool isCodecSpecific = false;
    if (msg.u.extended_buffer_data.flags & OMX_BUFFERFLAG_CODECCONFIG) {
        buffer->meta_data()->setInt32(kKeyIsCodecConfig, true);
        isCodecSpecific = true;
    }

    if (isGraphicBuffer || mQuirks & kOutputBuffersAreUnreadable) {
        buffer->meta_data()->setInt32(kKeyIsUnreadable, true);
    }

    buffer->meta_data()->setPointer(
            kKeyPlatformPrivate,
            msg.u.extended_buffer_data.platform_private);

    buffer->meta_data()->setPointer(
            kKeyBufferID,
            msg.u.extended_buffer_data.buffer);

    if (msg.u.extended_buffer_data.flags & OMX_BUFFERFLAG_EOS) {
        CODEC_LOGV("No more output data.");
        mNoMoreOutputData = true;
    }

    if (mIsEncoder && mIsVideo) {
        int64_t decodingTimeUs = isCodecSpecific? 0: getDecodingTimeUs();
        buffer->meta_data()->setInt64(kKeyDecodingTime, decodingTimeUs);
    }

    if (mTargetTimeUs >= 0) {
        CHECK(msg.u.extended_buffer_data.timestamp <= mTargetTimeUs);

        if (msg.u.extended_buffer_data.timestamp < mTargetTimeUs) {
            CODEC_LOGV(
                    "skipping output buffer at timestamp %lld us",
```

```
                                msg.u.extended_buffer_data.timestamp);

                        fillOutputBuffer(info);
                        break;
                    }

                    CODEC_LOGV(
                            "returning output buffer at target timestamp "
                            "%lld us",
                            msg.u.extended_buffer_data.timestamp);

                    mTargetTimeUs = -1;
                }

                mFilledBuffers.push_back(i);
                mBufferFilled.signal();
                if (mIsEncoder) {
                    sched_yield();
                }
            }

            break;
        }

        default:
        {
            CHECK(!"should not be here.");
            break;
        }
    }
}
```

当 OMXCodec 收到 FILL_BUFFER_DONE 后，将解码后的资料放入 mFilledBuffers，然后发出 mBufferFilled 信号，并要求 decoder 继续发出资料。

（4）使用函数 read()等待 mBufferFilled 信号，当 mFilledBuffers 被填入资料后，函数 read()将其指定给 Buffer，并回传给 AwesomePlayer。对应的实现代码如下所示。

```
status_t OMXCodec::read(
        MediaBuffer **buffer, const ReadOptions *options) {
…

    while (mState != ERROR && !mNoMoreOutputData && mFilledBuffers.empty()) {
        if ((err = waitForBufferFilled_l()) != OK) {
            return err;
        }
    }

    if (mState == ERROR) {
        return UNKNOWN_ERROR;
    }
```

```
    if (mFilledBuffers.empty()) {
        return mSignalledEOS ? mFinalStatus : ERROR_END_OF_STREAM;
    }

    if (mOutputPortSettingsHaveChanged) {
        mOutputPortSettingsHaveChanged = false;

        return INFO_FORMAT_CHANGED;
    }

    size_t index = *mFilledBuffers.begin();
    mFilledBuffers.erase(mFilledBuffers.begin());

    BufferInfo *info = &mPortBuffers[kPortIndexOutput].editItemAt(index);
    CHECK_EQ((int)info->mStatus, (int)OWNED_BY_US);
    info->mStatus = OWNED_BY_CLIENT;

    info->mMediaBuffer->add_ref();
    if (mSkipCutBuffer != NULL) {
        mSkipCutBuffer->submit(info->mMediaBuffer);
    }
    *buffer = info->mMediaBuffer;

    return OK;
}
```

函数 AwesomePlayer::onVideoEvent()除了通过 OMXCodec::read 取得解码后的资料外，还需要将这些资料（mVideoBuffer）传给 video renderer 以便在屏幕上显示。此功能的具体实现过程如下所示。

（1）在将 mVideoBuffer 中的资料输出之前，必须先建立 mVideoRenderer。对应的实现代码如下所示。

```
void AwesomePlayer::onVideoEvent()
{
  …
  if (mVideoRenderer == NULL)
  {
    initRenderer_l();
  }
  …
}
void AwesomePlayer::initRenderer_l()
{
  if (!strncmp("OMX.", component, 4))
  {
    mVideoRenderer = new AwesomeRemoteRenderer(
                             mClient.interface()->createRenderer(
                                                   mISurface,
                                                   component,
                                                   …));
```



```
  }
  else
  {
    mVideoRenderer = new AwesomeLocalRenderer(
                                    ...
                                    component,
                                    mISurface);
  }
}
```

（2）如果 video decoder 是 OMX component，则需要建立一个 AwesomeRemoteRenderer 作为 mVideoRenderer。从步骤（1）中的代码看，AwesomeRemoteRenderer 的核心功能是由函数 OMX::createRenderer()实现的。函数 createRenderer()先建立一个 hardware renderer—SharedVideoRenderer (libstagefrighthw.so)流程，如果失败则建立 software renderer—SoftwareRenderer(surface)流程。对应的实现代码如下所示。

```
sp<IOMXRenderer> OMX::createRenderer(...)
{
  VideoRenderer *impl = NULL;
  libHandle = dlopen("libstagefrighthw.so", RTLD_NOW);
  if (libHandle)
  {
    CreateRendererFunc func = dlsym(libHandle, ...);
    impl = (*func)(...);
  }
  if (!impl)
  {
    impl = new SoftwareRenderer(...);
  }
}
```

（3）如果 video decoder 是 software component，则需要建立一个 AwesomeLocalRenderer 作为 mVideoRenderer。AwesomeLocalRenderer 的 constructor 会呼叫本身的函数 init()，其具体功能和函数 OMX::createRenderer()的功能相同。对应的实现代码如下所示。

```
void AwesomeLocalRenderer::init(...)
{
  mLibHandle = dlopen("libstagefrighthw.so", RTLD_NOW);
  if (mLibHandle)
  {
    CreateRendererFunc func = dlsym(...);
    mTarget = (*func)(...);
  }
  if (mTarget == NULL)
  {
    mTarget = new SoftwareRenderer(...);
  }
}
```

（4）建立 mVideoRenderer 后就可以开始将解码后的资料回传，对应的实现代码如下所示。

```
void AwesomePlayer::onVideoEvent()
{
   if (!mVideoBuffer)
   {
      mVideoSource->read(&mVideoBuffer, ...);
   }
   [Check Timestamp]
   if (mVideoRenderer == NULL)
   {
      initRenderer_l();
   }
   mVideoRenderer->render(mVideoBuffer);
}
```

经过上述操作之后，Renderer 的处理过程介绍完毕。在播放多媒体时，需要使用 audio 来实现处理功能。在 StageFright 框架中，audio 的部分内容是由 AudioPlayer 来处理的，在函数 AwesomePlayer::play_l() 中被建立。下面将介绍使用 audio 的基本流程。

（1）当要求播放影音时，会同时建立并启动 AudioPlayer。对应的实现代码如下所示。

```
status_t AwesomePlayer::play_l()
{
   …
   mAudioPlayer = new AudioPlayer(mAudioSink, ...);
   mAudioPlayer->start(...);
   …
}
```

（2）在启动 AudioPlayer 的过程中会先读取第一笔解码后的资料，并开启 Audio Output。对应的实现代码如下所示。

```
status_t AudioPlayer::start(...)
{
   mSource->read(&mFirstBuffer);
   if (mAudioSink.get() != NULL)
   {
      mAudioSink->open(..., &AudioPlayer::AudioSinkCallback, ...);
      mAudioSink->start();
   }
   else
   {
      mAudioTrack = new AudioTrack(..., &AudioPlayer::AudioCallback, ...);
      mAudioTrack->start();
   }
}
```

在上述代码中，AudioPlayer 并没有将 mFirstBuffer 传给 Audio Output。

（3）在开启 Audio Output 的同时，AudioPlayer 将启用函数 callback()，这样每当函数 callback()被呼叫时 AudioPlayer 会去 audio decoder 读取解码后的资料。对应的实现代码如下所示。

```
size_t AudioPlayer::AudioSinkCallback(audioSink, buffer, size, ...)
{
   return fillBuffer(buffer, size);
}
void AudioPlayer::AudioCallback(..., info)
{
   buffer = info;
   fillBuffer(buffer->raw, buffer->size);
}
size_t AudioPlayer::fillBuffer(data, size)
{
   mSource->read(&mInputBuffer, ...);
   memcpy(data, mInputBuffer->data(), ...);
}
```

由上述代码可以知道，读取解码后的 Audio 资料的工作是由函数 callback()所驱动的，fillBuffer 会将资料（mInputBuffer）复制到数据 data 后，Audio Output 回去取用 data。

第 14 章　音频系统框架架构详解

在 Android 音频系统中，对应的硬件设备有音频输入和音频输出两部分。在手机设备中，输入设备通常是话筒，而输出设备通常是耳机和扬声器。Android 音频系统的核心是 Audio 系统，它在 Android 中负责音频方面的数据流传输和控制功能，也负责音频设备的管理。Audio 部分作为 Android 的 Audio 系统的输入/输出层次，一般负责 PCM 声音输出和从外部获取 PCM 声音，以及管理声音设备和设置。本章将详细讲解 Android 5.0 系统中音频系统框架的核心架构知识，为读者学习本书后面的知识打下基础。

14.1　硬件架构的发展趋势

在了解 Android 音频系统的核心架构之前，需要先了解硬件架构的发展历程。本节将以当今 Android 旗舰机 NOTE 3 的音频芯片高通 WCD9320 为素材进行剖析。

14.1.1　原始架构模式

音频芯片的发展也是一个相对漫长的过程，最初移动芯片组的集成度很低，处理音频的 CODEC 自然也是独立芯片，因为当时处理器性能太差，甚至还有专门用于多媒体处理器的 DSP 芯片，例如，海思处理器还叫 K3 时就是如此架构的。随着半导体产业的进步，移动处理器设计为了提高集成度和降低成本，往往把音频 CODEC 集成到 PMIC（电源管理 IC，类似联发科），甚至是 SoC 主芯片中，全志、瑞芯微的 ARM 等国内厂商的处理器正是这类设计。

在高通的早期处理器 MSM6 系、7 系产品中，也是将音频 CODEC 集成于处理器中，这样在一定程度上节省了外围电路设计，但是同时弊端也非常明显，主要有如下几条。

- ☑　芯片体积的限制使得音频的硬件 DSP 功能有限。
- ☑　音频处理更依赖于处理器的运算能力。
- ☑　音乐播放、系统混音等多媒体功能需要处理器频繁介入运算。
- ☑　ARM 处理器性能有限导致效果不好而且更为耗电。

另外，高集成度对音质的影响很大，这也是早期高通处理器智能手机平板产品、目前各类国产低价平板芯片组音质恶劣的首要原因。但对于需要快速使产品上市的手机厂商而言，设计方案的便利性很重要。所以超高集成和完善方案曾经一度占据上风，也是市场表现本不错的 TI 在联发科、高通面前逐渐败退，直至退出移动市场的原因。

14.1.2　移动处理器的解决方案

当从 ARM 处理器走向四核甚至八核开始，市场情况又出现了变化。在半导体工艺进步幅度有限

时，市场对移动处理器性能的需求呈爆发式增长，CPU、图形单元占据的面积越来越大，音频 CODEC 部分又被高通从 SoC 中剥离出来。随着高通 APQ8064 四核处理器的上市，其首个独立音频 CODEC 芯片 WCD9310（代号 TABLA）也逐渐被熟知。虽然经过十余款手机、平板电脑的音质测评证明，WCD9310 的音质并不算无敌，在海内外各类 IT 媒体的眼中这颗小芯片更是毫无存在感。但是它对于 Android 系统技术的进步，特别是音频子系统的进步起到了关键性作用。WCD9310+高通四核处理器带来的核心驱动和硬件设计一次性解决了被 Soomal 提出并困扰高通芯片组+Android 组合智能手机的难题：系统 SRC 和硬件 SRC 问题。

Android 系统的 SRC 问题在 3 年前还十分常见，Android 系统由于底层语言的问题，在音频播放上存在一个漏洞，即 48kHz 采样率转换为 44.1kHz 会生成劣质 SRC。这种劣质 SRC 的问题，使得音频信号在安卓设备里产生了扭曲和损耗，产生大量噪波，立体声播放层次等这些指标全面受损。而用户需求较多的高品质音乐母生带、高清视频、游戏等的音频都是高于 44.1kHz 的采样率，因此有很多用户抱怨安卓机器的音质失真和受损问题。因为音乐是重要的移动应用之一，智能手机的出现就是要替代主流的便携式播放器，智能手机、平板电脑中，没任何理由把音质做得比百元级别的随身听还要差。基于以上市场需求，解决 SRC 问题变得十分迫切。解决 SRC 的另一个重要原因则是为了更省电，系统自适应音频采样率播放，仅需要改变一下硬件驱动参数，而常见的 44.1kHz 和 48kHz 之间的重采样则是非整数倍的，非整数倍 SRC 需要一套数学算法实现，无论是系统 SRC 或硬件 SRC，无论算法的简单或复杂，SRC 都将意味着暴力的浮点运算，处理器资源和宝贵的电量被消耗在这种毫无意义的工作上。至今某些认为音质对普通人不重要、毫不在乎音质的开发者或厂商们请注意，只要智能手机中存在视频、游戏、铃声、通话等与音频相关的应用，都可能会触及 SRC 问题，这也意味着电量会更快耗完。虽然 Android SRC 时至今日也未能彻底消灭，但是主流的智能手机芯片组已经基本解决了这个问题，例如，谷歌产品 Nexus 5 的主板，其背面如图 14-1 所示。

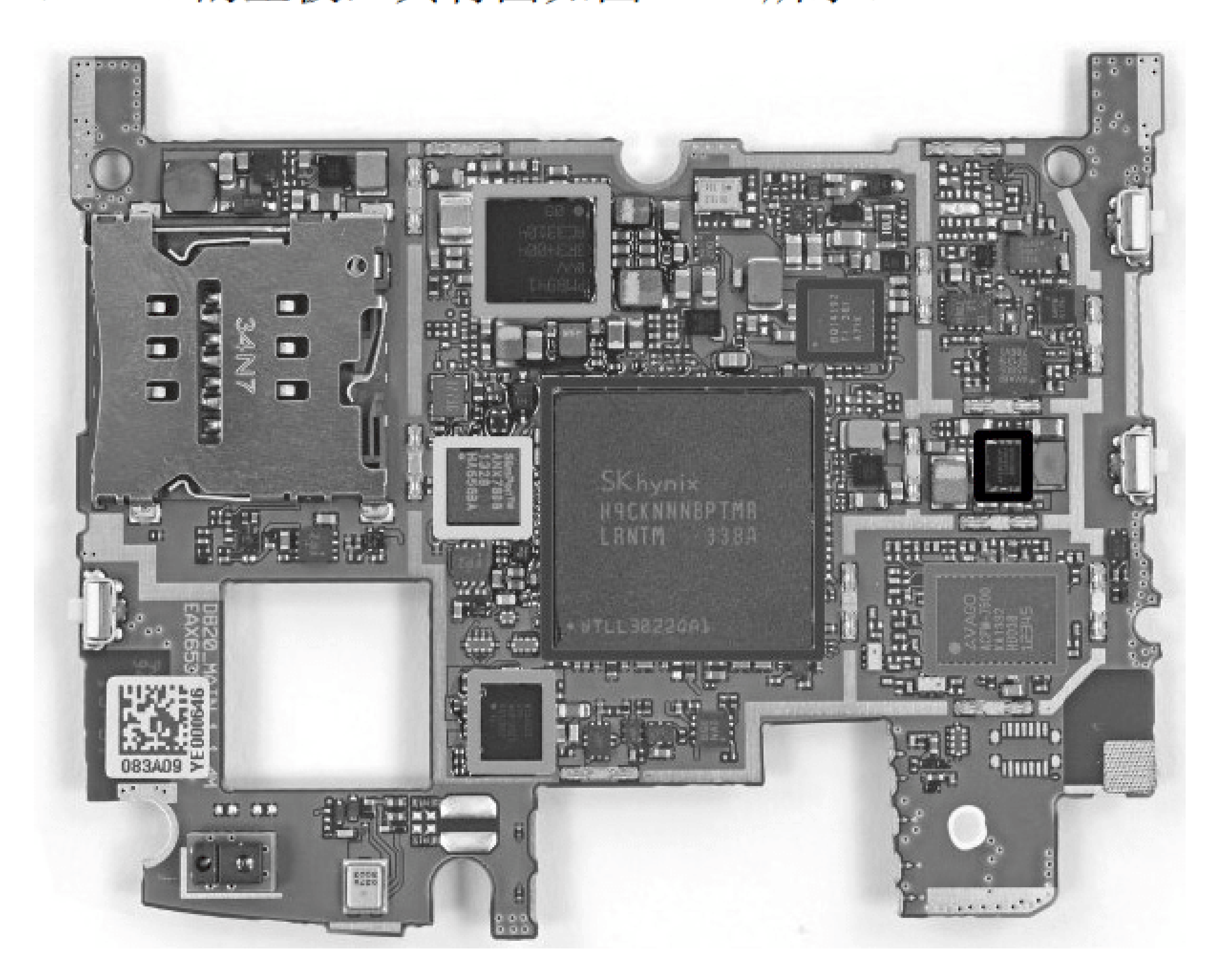


图 14-1　谷歌产品 Nexus 5 的硬件架构

14.1.3　升级版高通骁龙 801

2013 年高通推出了骁龙 800 处理器，也同样搭配了一颗新型号音频 CODEC 芯片 WCD9320（代号

TAIKO）。在巴塞罗那世界移动通信大会 MWC2014 上，高通公布了旗舰级移动处理器的小改款骁龙 801，如无意外搭配，CODEC 并不会出现变化。相比前一代 WCD9310，WCD9320 的进步和变化如下所示。

（1）低功耗播放

功耗控制依然是移动处理器的重点，Assertive Display 等技术的集成就是为了在处理器频率提升时依然能通过新技术进行省电控制。高通骁龙 800 还集成了超低功耗的硬件 DSP 处理器 Hexagon DSP6V5A，搭配了新 CODEC 和 Android 4.4 系统，可实现超长时间音乐播放。在使用耳机正常音量的情况下，普通手机音乐播放续航普遍在 20 小时左右，而 Google Nexus 5 可达 60 小时，实测时间可达到两天。这也意味着视频播放、语音通话等音频相关应用同样可以更省电。但是目前除 Nexus 5 外，绝大多数采用骁龙 800 处理器的手机、平板电脑官方固件还未升级至 Android 4.4 版本，因此暂时还无法感受这个优势。

除了省电之外，更多的用户关心的是新 CODEC 的音质表现，毕竟不是所有手机厂商都愿意使用独立 DAC 的设计，CODEC 素质基本决定了手机的音质表现。笔者用当前市场中主流机器测试的结果如表 14-1 所示。

表 14-1　CODEC 音质测试

测试项目[44.1kHz]	金立 E7	Google Nexus 5	Lumia 1520	索尼 L39h
噪声水平，dB（A）：	−87.0	−91.7	−90.1	−93.8
动态范围，dB（A）：	86.9	92.0	91.0	93.3
总谐波失真，%：	0.0057	0.0010	0.0091	0.0032
互调失真，%：	0.013	0.0087	0.015	0.0077
立体声分离度，dB：	−86.6	−93.4	−91.3	−92.7

从表 14-1 中手机的实际音质表现来看，高通 WCD9320 有着不错的指标，但缺点是最大不失真输出电平仅在-20dB 的水平，稍微提升一点都会导致严重的失真，耳机输出驱动力和动态必然会受到影响，这或许是“低功耗”带来的相应代价，而其中一个异类则是 WP 系统的 Lumia 1520，它的声音表现和其他几款 Android 手机差异较大，很可能采用了特殊的硬件设计。如果不考虑价格差异，观察上一代联发科与高通芯片组多款手机的竞争，会发现联发科阵营的产品整体音质是不落下风的，是否意味着 WCD9320 的优势更小？但事实是联发科的集成 CODEC 方案似乎也存在类似问题。

（2）支持高清音频

2013 年发布了两款采用高通骁龙 800 处理器的手机，分别是 LG G2 及三星 Galaxy Note 3，均号称支持最高 24bit/192kHz 采样率的高品质音乐回放。这是否意味着 Android 系统将会很快全面进化至高清音频平台？Android 是一个开放的系统，对于底层音频驱动的修改技术上可行，而硬件 CODEC 本身支持高采样音频也不是难题。但目前 Android 系统本身尚未支持 24bit 音频播放，LG 和三星恐怕并非通过硬件支持，而是通过软件转码降低采样实现的。

（3）提升语音通话质量

语音通话同样是音频 CODEC 负责的工作，有不少厂商选择了 Audience 的语音增强芯片增强通话质量，如 MI2S、三星 i9500 等，但是在使用高通骁龙 800 处理器的手机产品中，绝大多数的通话表现普遍更加优异。CODEC 硬件算法、驱动优化的进步同样改善了语音通话品质和降噪，Nexus 5、金立 E7、索尼 Xperia Z1 L39h 等型号在通话降噪测试中都有着不错的表现。



纵观当今硬件市场，究竟高通音频硬件架构的发展趋势是好还是坏？其实对于音质上无要求的手机用户来说，减少耗电自然是最有必要的。而对于有追求的手机厂商来说，提升耳机输出驱动力和音质也并非无解的难题。在国内以 vivo Xplay/Xplay 3S 为代表的“独立 DAC 芯片+高级运放”则代表了追求便携和音质的用户需求。因为手机硬件同质化越来越严重和 vivo 在一定范围内的成功，在现在和未来必然会有越来越多的国产手机品牌开始重视音质，这对广大消费者来说是一个好消息。

注意：首个解决 Android SRC 问题的厂商是步步高，解决方案是其旗下的 vivo 品牌公布的 VRS（vivo signal-Retrieval System）技术。

14.2　音频系统基础

Android 官方给出的音频系统的具体架构图如图 14-2 所示。

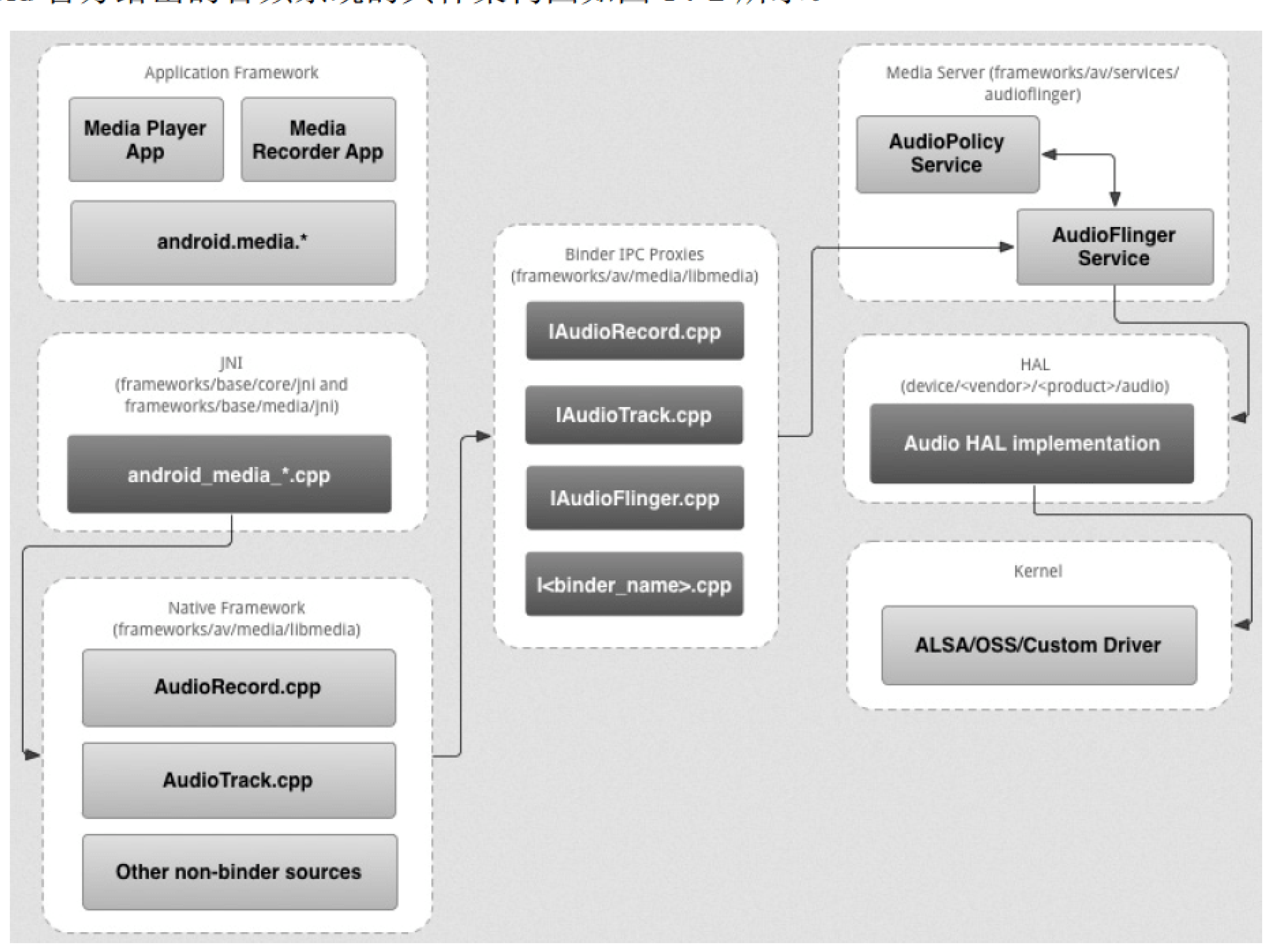


图 14-2　Android 官方给出的音频系统架构图

在 Audio 系统中，整个音频管理模块主要分成如下 4 个层次。

（1）Media 库提供的 Audio 系统本地部分接口。

（2）AudioFlinger 作为 Audio 系统的中间层。

（3）Audio 的硬件抽象层提供底层支持。

（4）Audio 接口通过 JNI 和 Java 框架提供给上层。

Android 音频系统的基本层次结构如图 14-3 所示。

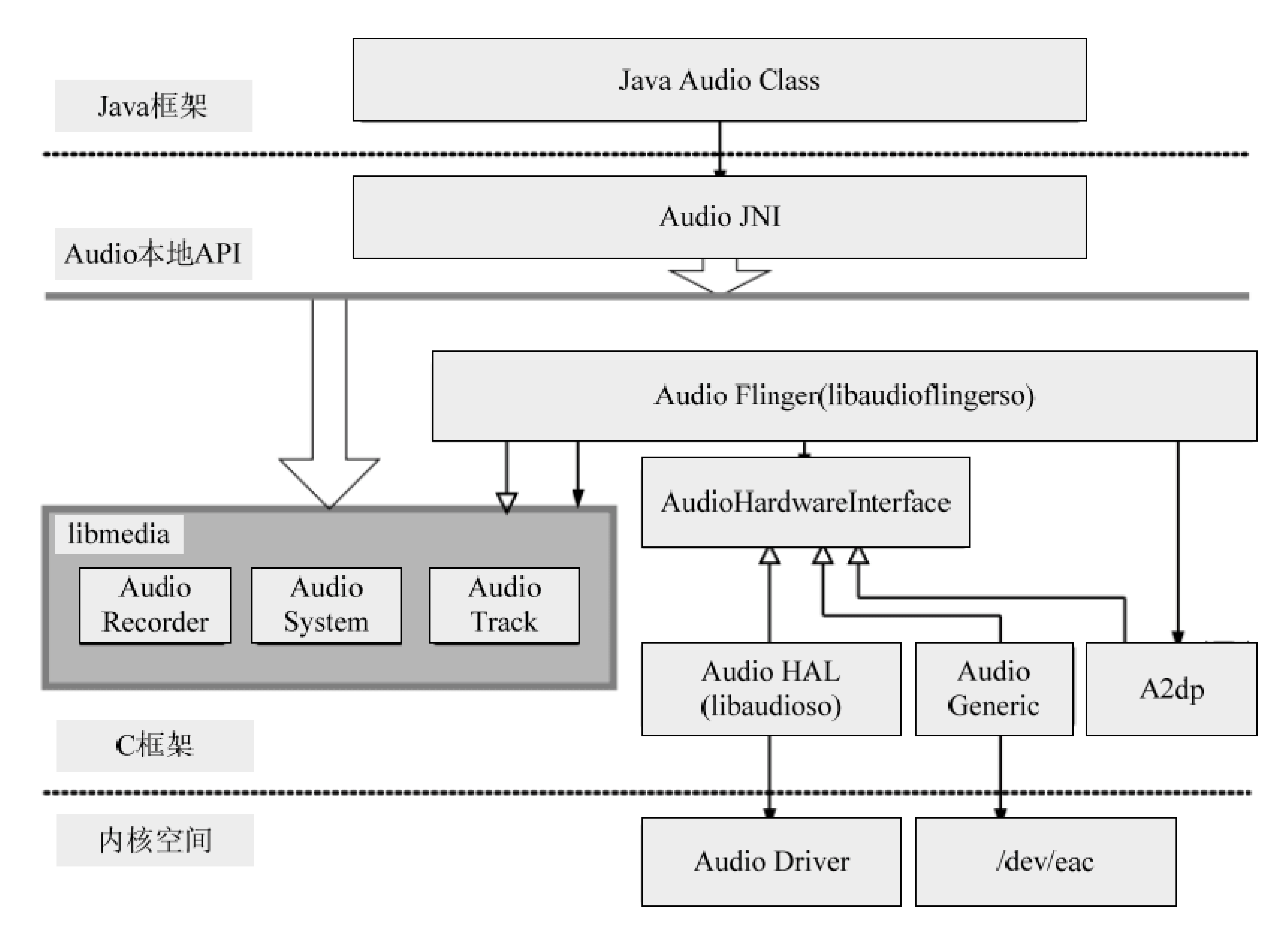


图 14-3　Android 音频系统的框架结构

图 14-3 中各个构成部分的具体说明如下所示。

（1）Audio 的 Java 部分

Java 部分的代码路径是 frameworks/base/media/java/android/media。

与 Audio 系统相关的 Java 包是 android.media，其中主要包含了与 AudioManager 和 Audio 系统等相关的类。

（2）Audio 的 JNI 部分

JNI 部分的代码路径是 frameworks/base/core/jni。

Audio 的 JNI 部分的生成库是 libandroid_runtime.so，Audio 的 JNI 是其中的一个部分。

（3）Audio 的框架部分

框架部分的头文件路径是 frameworks/base/include/media/。

具体实现源代码路径是 frameworks/base/media/libmedia/。

Audio 本地框架是 Media 库的一部分，本部分内容被编译成库 libmedia.so，提供 Audio 部分的接口（包括基于 Binder 的 IPC 机制）。

（4）Audio Flinger

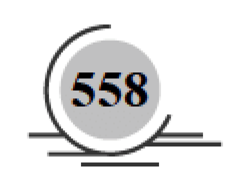
Flinger 部分的代码路径是 frameworks/base/libs/audioflinger。

Flinger 部分的内容被编译成库 libaudioflinger.so，这是 Audio 系统的本地服务部分。

（5）Audio 的硬件抽象层接口

硬件抽象层接口的头文件路径是 hardware/libhardware_legacy/include/hardware/。

在各个系统中，Audio 硬件抽象层的具体实现可能是不同的，需要使用代码去继承相应的类并实现它们，作为 Android 系统本地框架层和驱动程序接口。



14.3　音频系统的层次

在 Android 中，Audio 系统从上到下分别由 Java 的 Audio 类、Audio 本地框架类、AudioFlinger 和 Audio 的硬件抽象层几个部分组成。本节将简要介绍上述几个层次的基本知识。

14.3.1　层次说明

在 Android 中，Audio 系统各个层次的具体说明如下所示。

（1）Audio 本地框架类：是 libmedia.so 的一个部分，这些 Audio 接口对上层提供接口，由下层的本地代码去实现。

（2）AudioFlinger：继承了 libmedia 中的接口，提供实现库 libaudioflinger.so。这部分内容没有自己的对外头文件，上层调用的只是 libmedia 本部分的接口，但实际调用的内容是 libaudioflinger.so。

（3）JNI：在 Audio 系统中，使用 JNI 和 Java 对上层提供接口，JNI 部分通过调用 libmedia 库提供的接口来实现。

（4）Audio 硬件抽象层：提供到硬件的接口，供 AudioFlinger 调用。Audio 的硬件抽象层实际上是各个平台开发过程中需要主要关注和独立完成的部分。

因为 Android 中的 Audio 系统不涉及编/解码环节，只负责上层系统和底层 Audio 硬件的交互，所以通常以 PCM 作为输入/输出格式。

在 Android 的 Audio 系统中，无论上层还是下层，都使用一个管理类和输入/输出类来表示整个 Audio 系统，输入/输出类负责数据通道。Audio 系统在各个层次之间的对应关系如表 14-2 所示。

表 14-2　Android 各个层次的对应关系

层 次 说 明	Audio 管理环节	Audio 输出	Audio 输入
Java 层	android.media AudioSystem	android.media AudioTrack	android.media AudioRecorder
本地框架层	AudioSystem	AudioTrack	AudioRecorder
AudioFlinger	IAudioFlinger	IAudioTrack	IAudioRecorder
硬件抽象层	AudioHardwareInterface	AudioStreamOut	AudioStreamIn

14.3.2　Media 库中的 Audio 框架

在 Media 库中提供了 Android 的 Audio 系统的核心框架，在库中实现了 AudioSystem、AudioTrack 和 AudioRecorder 这 3 个类。另外还提供了 IAudioFlinger 类接口，通过此类可以获得 IAudioTrack 和 IAudioRecorder 两个接口，分别用于声音的播放和录制功能。AudioTrack 和 AudioRecorder 分别通过调用 IAudioTrack 和 IAudioRecorder 来实现。

Audio 系统的头文件被保存在 frameworks/av/include/media 目录中，其中包含的主要头文件如下所示。

☑　AudioSystem.h：Media 库的 Audio 部分对上层的总管接口。

☑　IAudioFlinger.h：需要下层实现的总管接口。

☑　AudioTrack.h：放音部分对上接口。

- ☑ IAudioTrack.h：放音部分需要下层实现的接口。
- ☑ AudioRecorder.h：录音部分对上接口。
- ☑ IAudioRecorder.h：录音部分需要下层实现的接口。

其中，文件 IAudioFlinger.h、IAudioTrack.h 和 IAudioRecorder.h 的接口是通过下层的继承来实现的。文件 AudioFlinger.h、AudioTrack.h 和 AudioRecorder.h 是对上层提供的接口，既供本地程序调用（例如声音的播放器、录制器等），也可以通过 JNI 向 Java 层提供接口。

从具体功能上看，AudioSystem 用于综合管理 Audio 系统，而 AudioTrack 和 AudioRecorder 负责分别输出和输入音频数据，即分别实现播放和录制功能。

AudioTrack 是 Audio 输出环节的类，其中包含了最重要的接口 write()，主要代码如下所示。

```
class AudioTrack : virtual public RefBase
{
public:
      enum channel_index {
            MONO = 0,
            LEFT = 0,
            RIGHT = 1
      };

      /* Events used by AudioTrack callback function (audio_track_cblk_t).
       * Keep in sync with frameworks/base/media/java/android/media/AudioTrack.java NATIVE_EVENT_*
       */
      enum event_type {
            EVENT_MORE_DATA = 0,



            EVENT_UNDERRUN = 1,
            EVENT_LOOP_END = 2,

            EVENT_MARKER = 3,

            EVENT_NEW_POS = 4,

            EVENT_BUFFER_END = 5
      };
{
      typedef void (*callback_t)(int event,
void* user, void *info);
      AudioTrack( int streamType,
                        uint32_t sampleRate = 0,          //音频的采样律
                        int format = 0,                   //音频的格式（例如 8 位或者 16 位的 PCM）
                        int channelCount = 0,             //音频的通道数
                        int frameCount = 0,               //音频的帧数
                        uint32_t flags = 0,
                        callback_t cbf = 0,
                        void* user = 0,
                        int notificationFrames = 0);
```



```
    void start();
    void stop();
    void flush();
    void pause();
    void mute(bool);
    ssize_t write(const void* buffer, size_t size);
…
        enum {
            NO_MORE_BUFFERS = 0x80000001, // same name in AudioFlinger.h, ok to be different value
            STOPPED = 1
        };
…
```

类 AudioRecord 用于实现和 Audio 录制相关的功能，其主要实现代码如下所示。

```
class AudioRecord
{
enum event_type {
    EVENT_MORE_DATA = 0,
    EVENT_OVERRUN = 1,
    EVENT_MARKER = 2,

    EVENT_NEW_POS = 3,

};
class Buffer
{
public:
    size_t        frameCount;



    size_t        size;
    union {
        void*         raw;
        short*        i16;
        int8_t*       i8;
    };
};
typedef void (*callback_t)(int event, void* user, void *info);
static status_t getMinFrameCount(size_t* frameCount,
                                  uint32_t sampleRate,
                                  audio_format_t format,
                                  audio_channel_mask_t channelMask);
```

在类 AudioTrack 和 AudioRecord 中，函数 read()和 write()的参数都是内存的指针及其大小，内存中的内容一般表示的是 Audio 的原始数据（PCM 数据）。这两个类还涉及 Audio 数据格式、通道数、帧数目等参数，可以在建立时指定，也可以在建立之后使用 set()函数进行设置。

另外，在 libmedia 库中提供的只是一个 Audio 系统框架，其中，类 AudioSystem、AudioTrack 和 AudioRecord 分别调用下层的接口 IAudioFlinger、IAudioTrack 和 IAudioRecord 来实现。另外的一个接

口是 IAudioFlingerClient，作为向 IAudioFlinger 中注册的监听器，相当于使用回调函数获取 IAudioFlinger 运行时的信息。

14.3.3 本地代码

在 Android 系统中，AudioFlinger 是 Audio 音频系统的中间层，能够作为 libmedia 提供的 Audio 部分接口的实现。这部分本地代码的路径为 frameworks/base/libs/audioflinger。

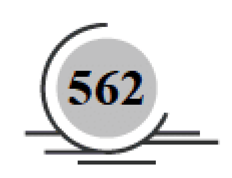
文件 AudioFlinger.h 和 AudioFlinger.cpp 是实现 AudioFlinger 的核心文件，其中提供了类 AudioFlinger，此类是一个 IAudioFlinger 的实现，其接口代码如下所示。

```
class AudioFlinger : public BnAudioFlinger,
public IBinder::DeathRecipient
{
public:
    static void instantiate();
    virtual status_t dump(int fd, const Vector<String16>& args);
    virtual sp<IAudioTrack> createTrack(
//获得音频输出接口（Track）
                                audio_stream_type_t streamType,
                                uint32_t sampleRate,
                                audio_format_t format,
                                audio_channel_mask_t channelMask,
                                size_t frameCount,
                                track_flags_t *flags,
                                const sp<IMemory>& sharedBuffer,
                                audio_io_handle_t output,
                                pid_t tid,
                                int *sessionId,
                                status_t *status) = 0;
//获得音频输出接口（Record）
    virtual sp<IAudioRecord> openRecord(
                                audio_io_handle_t input,
                                uint32_t sampleRate,
                                audio_format_t format,
                                audio_channel_mask_t channelMask,
                                size_t frameCount,
                                track_flags_t flags,
                                pid_t tid,
                                int *sessionId,
                                status_t *status) = 0;
```

由上述代码可以看出，AudioFlinger 使用函数 createTrack()来创建音频的输出设备 IAudioTrack，使用函数 openRecord()来创建音频的输入设备 IAudioRecord，并且还使用接口 get/set 来实现控制功能。

构造函数 AudioFlinger()的代码如下所示。

```
AudioFlinger::AudioFlinger()
{
    mHardwareStatus = AUDIO_HW_IDLE;
```



```
    mAudioHardware = AudioHardwareInterface::create();
    mHardwareStatus = AUDIO_HW_INIT;
    if (mAudioHardware->initCheck() == NO_ERROR) {
        mHardwareStatus = AUDIO_HW_OUTPUT_OPEN;
        status_t status;
        AudioStreamOut *hwOutput =
            mAudioHardware->openOutputStream (AudioSystem::PCM_16_BIT, 0, 0, &status);
        mHardwareStatus = AUDIO_HW_IDLE;
        if (hwOutput) {
            mHardwareMixerThread =
            new MixerThread(this, hwOutput, AudioSystem::AUDIO_OUTPUT_HARDWARE);
        } else {
            LOGE("Failed to initialize hardware   output stream, status: %d", status);
        }
    #ifdef WITH_A2DP
        mA2dpAudioInterface = new A2dpAudioInterface();
        AudioStreamOut *a2dpOutput = mA2dpAudioInterface->openOutputStream(AudioSystem::PCM_16_
BIT, 0, 0, &status);
        if (a2dpOutput) {
            mA2dpMixerThread = new MixerThread(this, a2dpOutput, AudioSystem::AUDIO_OUTPUT_A2DP);
            if (hwOutput) {
                uint32_t frameCount = ((a2dpOutput->bufferSize()/a2dpOutput->frameSize()) * hwOutput->
sampleRate()) / a2dpOutput->sampleRate();
                MixerThread::OutputTrack *a2dpOutTrack = new MixerThread::OutputTrack(mA2dpMixerThread,
                hwOutput->sampleRate(),
                AudioSystem::PCM_16_BIT,
                hwOutput->channelCount(),
                frameCount);
                mHardwareMixerThread->setOuputTrack(a2dpOutTrack);
            }
        } else {
            LOGE("Failed to initialize A2DP output stream, status: %d", status);
        }
#endif
        setRouting(AudioSystem::MODE_NORMAL, AudioSystem::ROUTE_SPEAKER, AudioSystem::ROUTE_
ALL);
        setRouting(AudioSystem::MODE_RINGTONE, AudioSystem::ROUTE_SPEAKER, AudioSystem::ROUTE_
ALL);
        setRouting(AudioSystem::MODE_IN_CALL, AudioSystem::ROUTE_EARPIECE, AudioSystem::ROUTE_
ALL);
        setMode(AudioSystem::MODE_NORMAL);
        setMasterVolume(1.0f);
        setMasterMute(false);
        mAudioRecordThread = new   AudioRecordThread(mAudioHardware, this);
        if (mAudioRecordThread != 0) {
            mAudioRecordThread->run("AudioRecordThread", PRIORITY_URGENT_AUDIO);
        }
      } else {
        LOGE("Couldn't even initialize the stubbed audio hardware!");
    }
}
```

由上述代码可以看出，在初始化 AudioFlinger 之后，会首先获得放音设备，然后为混音器（Mixer）建立线程，再建立放音设备线程，最后在线程中获得放音设备。

在文件 frameworks/av/services/audioflinger/AudioResampler.h 中定义了类 AudioResampler，此类是一个音频重取样器的工具类，定义代码如下所示。

```
class AudioResampler {
public:
    enum src_quality {
        DEFAULT=0,
        LOW_QUALITY=1,              //线性差值算法
        MED_QUALITY=2,              //立方差值算法
        HIGH_QUALITY=3              //fixed multi-tap FIR 算法
        VERY_HIGH_QUALITY=4,
    };
    static AudioResampler* create(int bitDepth, int inChannelCount,
            int32_t sampleRate, src_quality quality=DEFAULT_QUALITY);
    virtual ~AudioResampler();
    virtual void init() = 0;
    virtual void setSampleRate(int32_t inSampleRate);
    virtual void setVolume(int16_t left, int16_t right);
    virtual void setLocalTimeFreq(uint64_t freq);

    virtual void setPTS(int64_t pts);

    virtual void resample(int32_t* out, size_t outFrameCount,
            AudioBufferProvider* provider) = 0;

    virtual void reset();
    virtual size_t getUnreleasedFrames() const { return mInputIndex; }

    src_quality getQuality() const { return mQuality; }
```

在上述音频重取样工具类中，包含了如下 4 种质量。

☑ 低等质量（LOW_QUALITY）：使用线性差值算法实现。

☑ 中等质量（MED_QUALITY）：使用立方差值算法实现。

☑ 高等质量（HIGH_ QUALITY）：使用 FIR（有限阶滤波器）实现。

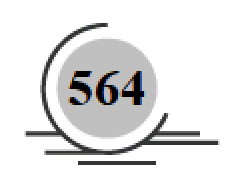
☑ VERY_HIGH_QUALITY：非常高质量，目前没有统一的实现标准。

在类 AudioResampler 中，AudioResamplerOrder1 是线性实现，AudioResamplerCubic.*文件提供立方实现方式，AudioResamplerSinc.*提供 FIR 实现。

通过文件 AudioMixer.h 和 AudioMixer.cpp 实现了一个 Audio 系统混音器，它被 AudioFlinger 调用，一般用于在声音输出之前的处理，提供多通道处理、声音缩放、重取样。AudioMixer 调用了 AudioResampler。

14.3.4 分析 JNI 代码

在 Android 的 Audio 系统中，通过 JNI 向 Java 层提供功能强大的接口，这样就可以在 Java 层通过 JNI 接口完成 Audio 系统的大部分操作。



Audio 系统 JNI 部分的实现代码被保存在 frameworks/base/core/jni 目录下，在此目录中主要有 3 个核心文件，分别对应了 Android Java 框架中 3 个类的支持，这 3 个文件的具体说明如下所示。

☑ android.media.AudioSystem：负责 Audio 系统的总体控制。

☑ android.media.AudioTrack：负责 Audio 系统的输出环节。

☑ android.media.AudioRecorder：负责 Audio 系统的输入环节。

在 Android 系统的 Java 层中，可以对 Audio 系统进行控制和数据流操作，其中控制操作和底层的处理基本一致；但是对于数据流操作，由于 Java 不支持指针，因此接口被封装成了另外的形式。例如，在音频输出功能中，通过文件 android_media_AudioTrack.cpp 提供了写字节和写短整型的接口类型。对应代码如下所示。

```
static jint android_media_AudioTrack_native_
write(JNIEnv *env,    jobject thiz,
jbyteArray javaAudioData,
jint offsetInBytes, jint sizeInBytes,
jint javaAudioFormat) {
      jbyte* cAudioData = NULL;
      AudioTrack *lpTrack = NULL;
      lpTrack = (AudioTrack *)env->GetIntField(
                        thiz, javaAudioTrackFields. Native TrackInJavaObj);
      ssize_t written = 0;
      if (lpTrack->sharedBuffer() == 0) {
      //进行写操作
            written = lpTrack->write(cAudioData +
offsetInBytes, sizeInBytes);
      } else {
            if (javaAudioFormat == javaAudioTrackFields.PCM16) {
                  memcpy(lpTrack->sharedBuffer()->pointer(),
                              cAudioData+offsetInBytes, sizeInBytes);
                  written = sizeInBytes;
            } else if (javaAudioFormat == javaAudioTrackFields.PCM8) {
                  int count = sizeInBytes;
                  int16_t *dst = (int16_t *)lpTrack->sharedBuffer()->pointer();
                  const int8_t *src = (const int8_t *)
(cAudioData + offsetInBytes);
                  while(count--) {
                        *dst++ = (int16_t)(*src++^0x80) << 8;
                  }
                  written = sizeInBytes;
            }
      }
      env->ReleasePrimitiveArrayCritical(javaAudioData, cAudioData, 0);
      return (int)written;
}
```

14.3.5 分析 Java 层代码

在 Android 的 Audio 系统中，和 Java 相关的类定义在包 android.media 中，Java 部分的代码保存在

frameworks/base/media/java/android/media 目录中，其中主要实现了如下类。

- ☑ android.media.AudioSystem
- ☑ android.media. AudioTrack
- ☑ android.media.AudioRecorder
- ☑ android.media.AudioFormat
- ☑ android.media.AudioService

其中，前 3 个类和本地代码是对应的，在 AudioFormat 中提供了一些和 Audio 相关的枚举值。在此需要注意的是，在 Audio 系统的 Java 代码中，虽然可以通过 AudioTrack 和 AudioRecorder 的 write()和 read()接口在 Java 层对 Audio 的数据流进行操作，但更多的时候并不需要这样做，而是在本地代码中直接调用接口进行数据流的输入/输出，而在 Java 层只进行控制类方面的操作，不处理具体的数据流。

（1）AudioSystem.java：提供了音频系统的基本类型定义，以及基本操作的接口。类 AudioSystem 对应于 JNI 层的文件 frameworks/base/core/jni/android_media_AudioSystem.cpp。主要实现代码如下所示。

```
public static final int PHONE_STATE_OFFCALL = 0;
public static final int PHONE_STATE_RINGING = 1;
public static final int PHONE_STATE_INCALL = 2;

public static final int FORCE_NONE = 0;
public static final int FORCE_SPEAKER = 1;
public static final int FORCE_HEADPHONES = 2;
public static final int FORCE_BT_SCO = 3;
public static final int FORCE_BT_A2DP = 4;
public static final int FORCE_WIRED_ACCESSORY = 5;
public static final int FORCE_BT_CAR_DOCK = 6;
public static final int FORCE_BT_DESK_DOCK = 7;
public static final int FORCE_ANALOG_DOCK = 8;
public static final int FORCE_DIGITAL_DOCK = 9;
public static final int FORCE_NO_BT_A2DP = 10;
public static final int FORCE_SYSTEM_ENFORCED = 11;
private static final int NUM_FORCE_CONFIG = 12;
public static final int FORCE_DEFAULT = FORCE_NONE;

public static final int FOR_COMMUNICATION = 0;
public static final int FOR_MEDIA = 1;
public static final int FOR_RECORD = 2;
public static final int FOR_DOCK = 3;
public static final int FOR_SYSTEM = 4;
private static final int NUM_FORCE_USE = 5;

public static final int SYNC_EVENT_NONE = 0;
public static final int SYNC_EVENT_PRESENTATION_COMPLETE = 1;

public static native int setDeviceConnectionState(int device, int state, String device_address);
public static native int getDeviceConnectionState(int device, String device_address);
public static native int setPhoneState(int state);
public static native int setForceUse(int usage, int config);
public static native int getForceUse(int usage);
```



```
public static native int initStreamVolume(int stream, int indexMin, int indexMax);
public static native int setStreamVolumeIndex(int stream, int index, int device);
public static native int getStreamVolumeIndex(int stream, int device);
public static native int setMasterVolume(float value);
public static native float getMasterVolume();
public static native int setMasterMute(boolean mute);
public static native boolean getMasterMute();
public static native int getDevicesForStream(int stream);

public static native int getPrimaryOutputSamplingRate();
public static native int getPrimaryOutputFrameCount();
public static native int getOutputLatency(int stream);

public static native int setLowRamDevice(boolean isLowRamDevice);
public static native int checkAudioFlinger();
```

（2）AudioService.java：此文件代表音频设置服务，在 SystemServer 中启动，功能是为所有的音频相关的设置提供服务。在 AudioService 中定义了一个 AudioSystemThread 的类，用来监控音频控制相关的信号，当有请求时，会通过调用 AudioSystem 的接口实现音频的控制，这里的消息处理是异步的。此外在 AudioService 中还抽象出了一套发送音频控制信号的接口为 AudioManager 提供支持。

（3）AudioManager.java：为上层应用提供了声音设置管理接口。

（4）Ringtone.java 和 RingtoneManager.java：功能是为铃声、闹钟等提醒提供了快速的播放以及管理接口。

（5）AudioTrack.java：功能是直接为 PCM 数据提供支持，在 JNI 层中的对应文件是 frameworks/base/core/jni/android_media_AudioTrack.cpp。

（6）SoundPool.java：提供了为引用播放声音的接口，在加载文件等方面做了优化。

（7）ToneGenerator.java：提供了播放 DTMF tones 的支持，可以直接为 PCM 数据提供支持，例如，电话的拨号音处理。此文件在 JNI 层中的对应文件是 frameworks/base/core/jni/android_media_ToneGenerator.cpp。

（8）AudioRecord.java：此文件是音频系统对外的录制接口，在 JNI 层中的对应文件是 frameworks/base/core/jni/android_media_AudioRecord.cpp。

14.4　Audio 系统的硬件抽象层

在 HAL 层定义了 Audio Service 调用的标准接口，不同的硬件必须根据自己的情况来实现这个接口让硬件在 Android 中正常工作，所以可以在不影响应用层系统调用的情况下更换不同的硬件，这样大大减少了系统耦合性。在 Android 5.0 系统中，Audio 硬件抽象层是 Audio 驱动程序和 Audio 本地框架类 AudioFlinger 的接口。根据 Android 系统对接口的定义，Audio 硬件抽象层是 C++类的接口，需要在继承接口中定义 3 个类来实现 Audio 硬件抽象层，这 3 个类分别实现总控、输入和输出功能。要想实现一个 Android 的硬件抽象层，则需要实现 AudioHardwareInterface、AudioStream Out 和 AudioStreamIn 这 3 个类，并将代码编译成动态库 libaudio.so。AudioFlinger 会连接这个动态库，并调用其中的 createAudioHardware()函数来获取接口。本节将详细讲解 Audio 系统的硬件抽象层的知识。

14.4.1 Audio 硬件抽象层基础

Audio 系统的硬件抽象层是 AudioFlinger 和 Audio 硬件之间的接口，在不同系统的移植过程中可以有不同的实现方式。其中，Audio 硬件抽象层的接口路径为 hardware/libhardware_legacy/include/hardware/。

在上述路径的核心文件是 AudioHardwareBase.h 和 AudioHardwareInterface.h。

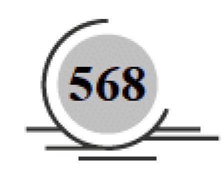
作为 Android 系统的 Audio 硬件抽象层，既可以用基于 Linux 标准的 ALSA 或 OSS 音频驱动来实现，也可以用基于私有的 Audio 驱动接口来实现。

在文件 AudioHardwareInterface.h 中，分别定义了类 AudioStreamOut、AudioStreamIn 和 AudioHardwareInterface。类 AudioStreamOut 和 AudioStreamIn 分别描述了音频输出设备和音频输入设备，其中负责数据流的接口分别是函数 write()和 read()，其参数是表示一块内存的指针和长度；另外还有一些设置和获取接口。类 AudioStreamOut 和 AudioStreamIn 的实现代码如下所示。

```
class AudioStreamOut {
public:
    virtual ~AudioStreamOut() = 0;
    virtual status_t setVolume(float volume) = 0;
    virtual ssize_t write(const void* buffer, size_t bytes) = 0;
    virtual int channelCount() const = 0;
virtual int format() const = 0;
virtual status_t setVolume(float volume) = 0;
virtual ssize_t write(const void* buffer, size_t bytes) = 0;
virtual status_t dump(int fd, const Vector<String16>& args) = 0;
};
class AudioStreamIn {
public:
    virtual ~AudioStreamIn() = 0;
    virtual status_t setGain(float gain) = 0;
    virtual ssize_t read(void* buffer, ssize_t bytes) = 0;
    virtual int channelCount() const = 0;
     virtual int format() const = 0;
    virtual status_t setGain(float gain) = 0;
    virtual ssize_t read(void* buffer, ssize_t bytes) = 0;
    virtual status_t dump(int fd, const Vector<String16>& args) = 0;
};
```

由此可见，类 AudioStreamOut 和 AudioStreamIn 是相互对应的接口类，分别实现输出和输入环节。类 AudioStreamOut 和 AudioStreamIn 都需要通过 Audio 硬件抽象层的核心 AudioHardwareInterface 接口类来获取。接口类 AudioHardwareInterface 的实现代码如下所示。

```
class AudioHardwareInterface {
public:
                            AudioHardwareInterface();
    virtual ~AudioHardwareInterface() { }
    virtual status_t initCheck() = 0;
    virtual status_t standby() = 0;
    virtual status_t setVoiceVolume(float volume) = 0;
    virtual status_t setMasterVolume(float volume) = 0;
```



```
    virtual status_t setRouting(int mode, uint32_t routes);
    virtual status_t getRouting(int mode, uint32_t* routes);
    virtual status_t setMode(int mode);
    virtual status_t getMode(int* mode);
    virtual status_t setMicMute(bool state) = 0;
    virtual status_t getMicMute(bool* state) = 0;
    virtual status_t setParameter(const char* key, const char* value);
    virtual AudioStreamOut* openOutputStream(                    //打开输出流
                                int format=0,
                                int channelCount=0,
                                uint32_t sampleRate=0) = 0;
    virtual AudioStreamIn* openInputStream(                      //打开输入流
                                int format,
                                int channelCount,
                                uint32_t sampleRate) = 0;
    virtual status_t dumpState(int fd, const Vector<String16>& args);
    static AudioHardwareInterface* create();
```

在上述 AudioHardwareInterface 接口的实现代码中，分别使用函数 openOutputStream()和 openInputStream()来获取类 AudioStreamOut 和类 AudioStreamIn，将其分别作为音频输出设备和输入设备来使用。

除此之外，在文件 AudioHardwareInterface.h 中还定义了 C 语言的接口来获取一个 AudioHardware Interface 类型的指针，具体的定义代码如下所示。

```
extern "C" AudioHardwareInterface* createAudioHardware(void);
```

14.4.2　AudioFlinger 中的 Audio 硬件抽象层的实现

在 Android 系统的 AudioFlinger 中，可以通过编译宏的方式来选择要使用的 Audio 硬件抽象层。可选择的 Audio 硬件抽象层既可以作为参考设计，也可以在没有实际的 Audio 硬件抽象层时使用，目的是保证系统的正常运行。

1．编译文件

文件 Android.mk 是 AudioFlinger 的编译文件，定义代码如下所示。

```
ifeq ($(strip $(BOARD_USES_GENERIC_AUDIO)),true)
  LOCAL_STATIC_LIBRARIES += libaudiointerface
else
  LOCAL_SHARED_LIBRARIES += libaudio
endif
LOCAL_MODULE:= libaudioflinger
include $(BUILD_SHARED_LIBRARY)
```

在上述代码中，当 BOARD_USES_GENERIC_AUDIO 为 True 时连接 libaudiointerface.a 静态库；当 BOARD_USES_GENERIC_AUDIO 为 False 时连接 libaudiointerface.so 动态库，在大多数情况下使用后者。

另外，在文件 Android.mk 中也生成了 libaudiointerface.a，具体代码如下所示。

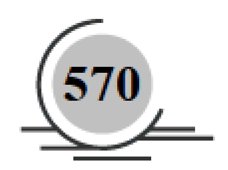
```
include $(CLEAR_VARS)
LOCAL_SRC_FILES:= \
      AudioHardwareGeneric.cpp \
      AudioHardwareStub.cpp \
      AudioDumpInterface.cpp \
      AudioHardwareInterface.cpp
LOCAL_SHARED_LIBRARIES := \
      libcutils \
      libutils \
      libmedia \
      libhardware_legacy
ifeq ($(strip $(BOARD_USES_GENERIC_AUDIO)),true)
   LOCAL_CFLAGS += -DGENERIC_AUDIO
endif
LOCAL_MODULE:= libaudiointerface
include $(BUILD_STATIC_LIBRARY)
```

在上述代码中，分别编译 4 个源文件来生成 libaudiointerface.a 静态库。其中，文件 AudioHardwareInterface.cpp 用于实现基础类和管理；文件 AudioHardwareGeneric.cpp、AudioHard wareStub.cpp 和 AudioDumpInterface.cpp 分别代表一种 Auido 硬件抽象层的实现，具体说明如下所示。

- ☑ AudioHardwareGeneric.cpp：实现基于特定驱动的通用 Audio 硬件抽象层。
- ☑ AudioHardwareStub.cpp：实现 Audio 硬件抽象层的一个桩。
- ☑ AudioDumpInterface.cpp：实现输出到文件的 Audio 硬件抽象层。

在文件 AudioHardwareInterface.cpp 中定义了 AudioHardwareInterface::create()函数，此函数是 Audio 硬件抽象层的创建函数，主要代码如下所示。

```
AudioHardwareInterface* AudioHardwareInterface::create()
{
      AudioHardwareInterface* hw = 0;
      char value[PROPERTY_VALUE_MAX];
#ifdef GENERIC_AUDIO
      hw = new AudioHardwareGeneric();
//此处用通用的 Audio 硬件抽象层
#else
      if (property_get("ro.kernel.qemu", value, 0)) {
            LOGD("Running in emulation - using generic audio driver");
            hw = new AudioHardwareGeneric();
      }
      else {
            LOGV("Creating Vendor Specific AudioHardware");
            hw = createAudioHardware();
//此处用实际的 Audio 硬件抽象层
      }
#endif
      if (hw->initCheck() != NO_ERROR) {
            LOGW("Using stubbed audio hardware.No sound will be produced.");
            delete hw;
            hw = new AudioHardwareStub();
//此处用实际的 Audio 硬件抽象层的桩实现
```



```
	}
#ifdef DUMP_FLINGER_OUT
	hw = new AudioDumpInterface(hw);
//此处用实际的 Audio 的 Dump 接口实现
#endif
	return hw;
}
```

2．桩方式实现

在文件 AudioHardwareStub.h 和 AudioHardwareStub.cpp 中，通过桩方式实现了一个 Android 硬件抽象层。桩方式不操作实际的硬件和文件，只是进行了空操作。当在系统中没有实际的 Audio 设备时才使用桩方式实现，目的是保证系统正常工作。如果使用这个硬件抽象层，实际上 Audio 系统的输入和输出都将为空。

在文件 AudioHardwareStub.h 中定义了类 AudioStreamOutStub 和类 AudioStreamInStub，分别实现输出和输入。主要实现代码如下所示。

```
class AudioStreamOutStub : public AudioStreamOut {
public:
	virtual status_t set(int format, int
channelCount, uint32_t sampleRate);
	virtual uint32_t sampleRate() const { return 44100; }
	virtual size_t bufferSize() const { return 4096; }
	virtual int channelCount() const { return 2; }
	virtual int format() const { return
  AudioSystem::PCM_16_BIT; }
	virtual uint32_t latency() const { return 0; }
	virtual status_t setVolume(float volume) { return NO_ERROR; }
	virtual ssize_t write(const void* buffer, size_t bytes);
	virtual status_t standby();
	virtual status_t dump(int fd, const Vector<String16>& args);
};
class AudioStreamInStub : public AudioStreamIn {
public:
	virtual status_t set(int format, int
  channelCount, uint32_t sampleRate, AudioSystem::
audio_in_acoustics acoustics);
	virtual uint32_t sampleRate() const { return 8000; }
	virtual size_t bufferSize() const { return 320; }
	virtual int channelCount() const { return 1; }
	virtual int format() const { return
  AudioSystem::PCM_16_BIT; }
	virtual status_t setGain(float gain) { return NO_ERROR; }
	virtual ssize_t read(void* buffer, ssize_t bytes);
	virtual status_t dump(int fd, const Vector<String16>& args);
	virtual status_t standby() { return NO_ERROR; }
};
```

在上述代码中，只用缓冲区大小、采样率和通道数这 3 个固定的参数将一些函数直接无错误返回，然后需要使用类 AudioHardwareStub 来继承类 AudioHardwareBase，也就是继承类 AudioHardwareInterface。

主要实现代码如下所示。

```
class AudioHardwareStub : public  AudioHardwareBase
{
public:
                        AudioHardwareStub();
    virtual ~AudioHardwareStub();
    virtual status_t initCheck();
    virtual status_t setVoiceVolume(float volume);
    virtual status_t setMasterVolume(float volume);
    virtual status_t setMicMute(bool state)
{ mMicMute = state;  return NO_ERROR; }
    virtual status_t getMicMute(bool* state)
{ *state = mMicMute ; return NO_ERROR; }
    virtual status_t setParameter(const
char* key, const char* value)
            { return NO_ERROR; }
    virtual AudioStreamOut* openOutputStream(          //打开输出流
                                int format=0,
                                int channelCount=0,
                                uint32_t sampleRate=0,
                                status_t *status=0);

    virtual AudioStreamIn* openInputStream(            //打开输入流
                                int format,
                                int channelCount,
                                uint32_t sampleRate,
                                status_t *status,
                                AudioSystem::audio_in_acoustics acoustics);
......
```

为了保证可以输入和输出声音，桩实现的主要内容是实现类 AudioStreamOutStub 和类 AudioStreamInStub 的读/写函数。主要实现代码如下所示。

```
ssize_t AudioStreamOutStub::write(const void* buffer, size_t bytes)
{
    usleep(bytes * 1000000 / sizeof(int16_t) /
channelCount() / sampleRate());
    return bytes;
}
ssize_t AudioStreamInStub::read(void* buffer, ssize_t bytes)
{
    usleep(bytes * 1000000 / sizeof(int16_t) /
channelCount() / sampleRate());
    memset(buffer, 0, bytes);
    return bytes;
}
```

当使用这个接口来输入和输出音频时，与真实的设备并没有任何关系，输出和输入都使用延时来完成。在输出时不会播出声音，但是返回值表示全部内容已经输出完成；在输入时会返回全部为 0 的数据。



3．通用 Audio 硬件抽象层

在 Android 系统中，文件 AudioHardwareGeneric.h 和 AudioHardwareGeneric.cpp 实现了通用的 Audio 硬件抽象层。与前面介绍的桩实现方式不同，这是一个真正能够使用的 Audio 硬件抽象层，但是它需要 Android 的一种特殊的声音驱动程序的支持。

在通用硬件抽象层中，类 AudioStreamOutGeneric、AudioStreamInGeneric 和 AudioHardwareGeneric 分别继承 Audio 硬件抽象层的 3 个接口。对应代码如下所示。

```
class AudioStreamOutGeneric : public AudioStreamOut {
    // ... 通用 Audio 输出类的接口
};
class AudioStreamInGeneric : public AudioStreamIn {
    // ... 通用 Audio 输入类的接口
};
class AudioHardwareGeneric : public AudioHardwareBase
{
    // ... 通用 Audio 控制类的接口
};
```

在文件 AudioHardwareGeneric.cpp 中使用的驱动程序是/dev/eac，这是一个非标准程序，定义设备路径的代码如下所示。

```
static char const * const kAudioDeviceName = "/dev/eac";
```

注意： eac 是 Linux 中的一个 misc 驱动程序，作为 Android 的通用音频驱动，写设备表示放音，读设备表示录音。

在 Linux 操作系统中，/dev/eac 驱动程序在文件系统中的节点主设备号为 10，此设备号是自动生成的。通过构造函数 AudioHardwareGeneric()可以打开这个驱动程序的设备节点。对应代码如下所示。

```
AudioHardwareGeneric::AudioHardwareGeneric()
    : mOutput(0), mInput(0), mFd(-1), mMicMute(false)
{
    mFd = ::open(kAudioDeviceName, O_RDWR);        //打开通用音频设备的节点
}
```

此音频设备是一个比较简单的驱动程序，其中并没有很多设置接口，只是用写设备来表示录音，用读设备来表示放音。放音和录音支持的都是 16 位的 PCM。对应的实现代码如下所示。

```
ssize_t AudioStreamOutGeneric::write(const void* buffer, size_t bytes)
{
    Mutex::Autolock _l(mLock);
    return ssize_t(::write(mFd, buffer, bytes));            //写入硬件设备
}
ssize_t AudioStreamInGeneric::read(void* buffer, ssize_t bytes)
{
    AutoMutex lock(mLock);
    if (mFd < 0) {
        return NO_INIT;
    }
```

```
    return ::read(mFd, buffer, bytes);                    //读取硬件设备
}
```

尽管 AudioHardwareGeneric 是一个可以真正工作的 Audio 硬件抽象层，但是这种实现方式非常简单，不支持各种设置，参数也只能使用默认的。而且这种驱动程序需要在 Linux 核心加入 eac 驱动程序的支持。

4．具备 Dump 功能的 Audio 硬件抽象层

在文件 AudioDumpInterface.h 和 AudioDumpInterface.cpp 中，提供了具备 Dump 功能的 Audio 硬件抽象层，目的是将输出的 Audio 数据写入到文件中。

其实 AudioDumpInterface 本身支持 Audio 输出功能，但是不支持输入功能。在文件 AudioDumpInterface.h 中定义类的代码如下所示。

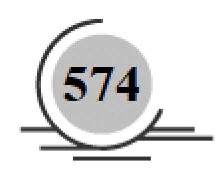
```
class AudioStreamOutDump : public AudioStreamOut {
public:
                              AudioStreamOutDump( AudioStreamOut* FinalStream);
                              ~AudioStreamOutDump();
                              virtual ssize_t write(const void* buffer, size_t bytes);
    virtual uint32_t sampleRate() const { return mFinalStream->sampleRate(); }
    virtual size_t bufferSize() const { return mFinalStream->bufferSize(); }
    virtual int channelCount() const { return
 mFinalStream->channelCount(); }
    virtual int format() const { return mFinalStream->format(); }
    virtual uint32_t latency() const { return    mFinalStream->latency(); }
    virtual status_t setVolume(float volume)
     { return    mFinalStream->setVolume(volume); }
    virtual status_t standby();
};
class AudioDumpInterface : public AudioHardwareBase
{
    virtual AudioStreamOut* openOutputStream(
                                      int format=0,
                                      int channelCount=0,
                                      uint32_t sampleRate=0,
                                      status_t *status=0);
}
```

在上述代码中，只实现了 AudioStreamOut 输出，而没有实现 AudioStreamIn 输入。由此可见，此 Audio 硬件抽象层只支持输出功能，不支持输入功能。其中，输出文件的名称被定义为如下格式。

```
#define FLINGER_DUMP_NAME "/data/FlingerOut.pcm"
```

在文件 AudioDumpInterface.cpp 中，通过函数 AudioStreamOut()实现写操作，写入的对象就是这个文件。对应的实现代码如下所示。

```
ssize_t AudioStreamOutDump::write(const void* buffer, size_t bytes)
{
    ssize_t ret;
    ret = mFinalStream->write(buffer, bytes);
    if(!mOutFile && gFirst) {
```



```
            gFirst = false;
            mOutFile = fopen(FLINGER_DUMP_NAME, "r");
            if(mOutFile) {
                fclose(mOutFile);
                mOutFile = fopen(FLINGER_DUMP_NAME, "ab");
//打开输出文件
            }
        }
        if (mOutFile) {
            fwrite(buffer, bytes, 1, mOutFile);
//写文件输出内容
        }
        return ret;
}
```

如果文件是打开的，则可以使用追加方式写入。当使用这个 Audio 硬件抽象层时，播放的内容（PCM）将全部被写入到文件。而且这个类支持各种格式的输出，具体格式将取决于调用者的设置。

使用 AudioDumpInterface 的目的并不是为了实际的应用，而是为了调试所使用的类。当使用播放器调试音频时，有时无法确认是解码器的问题还是 Audio 输出单元的问题，这时就可以用这个类来替换实际的 Audio 硬件抽象层，将解码器输出的 Audio 的 PCM 数据写入文件中，由此可以判断解码器的输出是否正确。

14.4.3　真正实现 Audio 硬件抽象层

要想实现一个真正的 Audio 硬件抽象层，需要完成和 14.2 节中实现硬件抽象层类似的工作。例如，可以基于 Linux 标准的音频驱动 OSS（Open Sound System）或 ALSA（Advanced Linux Sound Architecture）驱动程序来实现。

（1）基于 OSS 驱动程序实现

对于 OSS 驱动程序来说，实现方式和前面的 AudioHardwareGeneric 方式类似，数据流的读/写操作通过对/dev/dsp 设备的“读/写”来完成，区别在于 OSS 支持了更多的 ioctl 来进行设置，还涉及通过/dev/mixer 设备进行控制，并支持更多不同的参数。

（2）ALSA 驱动程序

对于 ALSA 驱动程序来说，实现方式一般不是直接调用驱动程序的设备节点，而是先实现用户空间的 alsa-lib，然后 Audio 硬件抽象层通过调用 alsa-lib 来实现。

在实现 Audio 硬件抽象层时，如果系统中有多个 Audio 设备，此时可由硬件抽象层自行处理，通过 setRouting()函数来设定。例如，可以选择支持多个设备的同时输出，或者有优先级输出。对于这种情况，数据流一般来自函数 AudioStreamOut::write()，可由硬件抽象层确定输出方法。对于某种特殊的情况，也有可能采用硬件直接连接的方式，此时数据流可能并不来自上面的 write()，这样就没有数据通道，只有控制接口。Audio 硬件抽象层也是可以处理这种情况的。

14.5　Kernel Driver 实现

在 Android 系统中，Audio 驱动负责与硬件进行交互，并且实现 HAL 层的接口供上层正常调用。

厂商可以选择使用 ALSA、OSS 以及自定义的音频驱动。如果选择 ALSA，Android 官方建议使用 external/tinyalsa 目录下的实现。下面将以切换音频通道为例（在通话过程中要将 Audio Output Path 从蓝牙耳机切换到 Speaker）讲解 Audio 系统的 Kernel Driver 实现过程。

在 Audio Path 的切换过程中，Android 提供了策略管理器来分配输入/输出的设备。例如，当手机播放音乐时默认从 Speaker 播放出来，这时若插入耳机，则会从耳机设备输出。但是有时想要自己去指定，例如在通话时打开了免提，实际上也就是将 Audio Path 切换到了 Speaker，即打开了外方喇叭。此时在代码中只需调用一个函数即可，这样只是强制切换 Audio Path，并没有遵从系统的分配。

```
AudioManager audioManager = (AudioManager) context.getSystemService(Context.AUDIO_SERVICE);
        audioManager.setSpeakerphoneOn(true);
```

最终调用 JNI 文件 android_media_AudioSystem 中的 android_media_AudioSystem_setForceUse()函数，具体实现代码如下所示。

```
static int
android_media_AudioSystem_setForceUse(JNIEnv *env, jobject thiz, jint usage, jint config)
{
    SLOGE("jni android_media_AudioSystem_setForceUse()");
    return  check_AudioSystem_Command(AudioSystem::setForceUse(static_cast  <audio_policy_force_use_
t>(usage),
                                                          static_cast
<audio_policy_forced_cfg_t>(config)));
}
```

在上述代码中需要重点注意 audio_policy_force_use_t 和 audio_policy_forced_cfg_t 这两个结构体，其中，audio_policy_force_use_t 表示当前的 Audio 环境，audio_policy_forced_cfg_t 表示 Audio 的输入/输出设备。这两个是专门为 setForceUse 所用的，具体代码如下所示。

```
/* usages used for audio_policy->set_force_use() */
typedef enum {
    //表示的是通话过程中
    AUDIO_POLICY_FORCE_FOR_COMMUNICATION,
    //媒体
    AUDIO_POLICY_FORCE_FOR_MEDIA,
    //录音
    AUDIO_POLICY_FORCE_FOR_RECORD,
    AUDIO_POLICY_FORCE_FOR_DOCK,
    AUDIO_POLICY_FORCE_FOR_SYSTEM,

    AUDIO_POLICY_FORCE_USE_CNT,
    AUDIO_POLICY_FORCE_USE_MAX = AUDIO_POLICY_FORCE_USE_CNT - 1,
} audio_policy_force_use_t;
/* device categories used for audio_policy->set_force_use() */
typedef enum {
    AUDIO_POLICY_FORCE_NONE,
    AUDIO_POLICY_FORCE_SPEAKER,
    AUDIO_POLICY_FORCE_HEADPHONES,
    AUDIO_POLICY_FORCE_BT_SCO,
    AUDIO_POLICY_FORCE_BT_A2DP,
```

```
    AUDIO_POLICY_FORCE_WIRED_ACCESSORY,
    AUDIO_POLICY_FORCE_BT_CAR_DOCK,
    AUDIO_POLICY_FORCE_BT_DESK_DOCK,
    AUDIO_POLICY_FORCE_ANALOG_DOCK,
    AUDIO_POLICY_FORCE_DIGITAL_DOCK,
    AUDIO_POLICY_FORCE_NO_BT_A2DP, /* A2DP sink is not preferred to speaker or wired HS */
    AUDIO_POLICY_FORCE_SYSTEM_ENFORCED,

    AUDIO_POLICY_FORCE_CFG_CNT,
    AUDIO_POLICY_FORCE_CFG_MAX = AUDIO_POLICY_FORCE_CFG_CNT - 1,

    AUDIO_POLICY_FORCE_DEFAULT = AUDIO_POLICY_FORCE_NONE,
} audio_policy_forced_cfg_t;
```

要想在通话时打开 Speaker，则传递的参数就是 usage 和 config，分别代表 AUDIO_POLICY_FORCE_FOR_COMMUNICATION 和 AUDIO_POLICY_FORCE_SPEAKER，这两个参数从上层一直传递到底层。

接着需要调用文件 AudioSystem.cpp 中的 setForceUse()函数，具体实现代码如下所示。

```
status_t AudioSystem::setForceUse(audio_policy_force_use_t usage, audio_policy_forced_cfg_t config)
{
    SLOGE("setForceUse() usage = %d,   config = %d" ,usage , config);
    const sp<IAudioPolicyService>& aps = AudioSystem::get_audio_policy_service();
    if (aps == 0) return PERMISSION_DENIED;
    return aps->setForceUse(usage, config);
}
```

函数 get_audio_policy_service()的功能是通过 Native 的 ServiceManager 来获取 audio policy 的 Service 代理对象的，从而实现与 audio policy 的进程间通信，对应代码如下所示。

```
…
binder = sm->getService(String16("media.audio_policy"));
…
```

接下来调用文件 frameworks/av/services/audioflinger/AudioPolicyService.cpp 中的函数 setForceUse()，具体实现代码如下所示。

```
status_t AudioPolicyService::setForceUse(audio_policy_force_use_t usage,
                                         audio_policy_forced_cfg_t config)
{
    if (mpAudioPolicy == NULL) {
        return NO_INIT;
    }
    if (!settingsAllowed()) {
        return PERMISSION_DENIED;
    }
    if (usage < 0 || usage >= AUDIO_POLICY_FORCE_USE_CNT) {
        return BAD_VALUE;
    }
    if (config < 0 || config >= AUDIO_POLICY_FORCE_CFG_CNT) {
        return BAD_VALUE;
    }
    Mutex::Autolock _l(mLock);
```

```
        mpAudioPolicy->set_force_use(mpAudioPolicy, usage, config);
        return NO_ERROR;
}
```

在上述代码中，mpAudioPolicy 的 set_force_use()函数在哪里实现呢？首先需要明白，mpAudioPolicy 是一个指针，在文件 AudioServicePolicy.cpp 的构造函数中被赋值，具体赋值过程如下所示。

```
…
const struct hw_module_t *module;
…
rc = hw_get_module(AUDIO_POLICY_HARDWARE_MODULE_ID, &module);
…
rc = audio_policy_dev_open(module, &mpAudioPolicyDev);
…
rc = mpAudioPolicyDev->create_audio_policy(mpAudioPolicyDev,&aps_ops,this, &mpAudioPolicy);
…
```

AUDIO_POLICY_HARDWARE_MODULE_ID 的值如下所示。

```
#define AUDIO_POLICY_HARDWARE_MODULE_ID "audio_policy"
```

module 是一个指针，指向的是一个 hw_module_t 结构体类型，其作用是调用系统的 audio policy module，这个 module 可以是原始的，也可以由厂商自定义实现。具体实现代码如下所示。

```
typedef struct hw_module_t {
/** tag must be initialized to HARDWARE_MODULE_TAG */
        uint32_t tag;
        uint16_t module_api_version;
        #define version_major module_api_version
        uint16_t hal_api_version;
#define version_minor hal_api_version

        /** Identifier of module */
        const char *id;

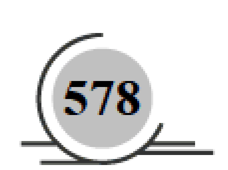
        /** Name of this module */
        const char *name;

        /** Author/owner/implementor of the module */
        const char *author;

        /** Modules methods */
        struct hw_module_methods_t* methods;

        /** module's dso */
        void* dso;

        /** padding to 128 bytes, reserved for future use */
        uint32_t reserved[32-7];
} hw_module_t;
```



在文件 hardware.c 中给 module 赋值的实现代码如下所示。

```
int hw_get_module(const char *id, const struct hw_module_t **module)
{
    return hw_get_module_by_class(id, NULL, module);
}
```

在文件 hardware.c 中，hw_get_module_by_class()方法的功能是找到指定的库文件并且加载，具体实现代码如下所示。

```
int hw_get_module_by_class(const char *class_id, const char *inst,
                           const struct hw_module_t **module)
{
    int status;
    int i;
    const struct hw_module_t *hmi = NULL;
    char prop[PATH_MAX];
    char path[PATH_MAX];
    char name[PATH_MAX];

    if (inst)
        snprintf(name, PATH_MAX, "%s.%s", class_id, inst);
    else
        strlcpy(name, class_id, PATH_MAX);

    /*
     * Here we rely on the fact that calling dlopen multiple times on
     * the same .so will simply increment a refcount (and not load
     * a new copy of the library).
     * We also assume that dlopen() is thread-safe
     */

    /* Loop through the configuration variants looking for a module */
    for (i=0 ; i<HAL_VARIANT_KEYS_COUNT+1 ; i++) {
        if (i < HAL_VARIANT_KEYS_COUNT) {
            if (property_get(variant_keys[i], prop, NULL) == 0) {
                continue;
            }
            snprintf(path, sizeof(path), "%s/%s.%s.so",
                     HAL_LIBRARY_PATH2, name, prop);
            if (access(path, R_OK) == 0) break;

            snprintf(path, sizeof(path), "%s/%s.%s.so",
                     HAL_LIBRARY_PATH1, name, prop);
            if (access(path, R_OK) == 0) break;
        } else {
            snprintf(path, sizeof(path), "%s/%s.default.so",
                     HAL_LIBRARY_PATH1, name);
            if (access(path, R_OK) == 0) break;
        }
    }
```

```
        status = -ENOENT;
        if (i < HAL_VARIANT_KEYS_COUNT+1) {
            /* load the module, if this fails, we're doomed, and we should not try
             * to load a different variant */
            status = load(class_id, path, module);
        }

        return status;
}
```

这样会得到库 audio_policy.default.so，这个库正是编译 hardware/libhardware_legacy/audio 得到的。再跳回到 AudioPolicyService 的构造函数中，接下来执行如下代码。

```
rc = audio_policy_dev_open(module, &mpAudioPolicyDev);
```

上述代码调用的是 legacy_ap_dev_open()函数，此函数在文件 audio_policy_hal.cpp 中实现，具体实现代码如下所示。

```
static int legacy_ap_dev_open(const hw_module_t* module, const char* name,
                                            hw_device_t** device)
{
        struct legacy_ap_device *dev;

        if (strcmp(name, AUDIO_POLICY_INTERFACE) != 0)
            return -EINVAL;

        dev = (struct legacy_ap_device *)calloc(1, sizeof(*dev));
        if (!dev)
            return -ENOMEM;

        dev->device.common.tag = HARDWARE_DEVICE_TAG;
        dev->device.common.version = 0;
        dev->device.common.module = const_cast<hw_module_t*>(module);
        dev->device.common.close = legacy_ap_dev_close;
        dev->device.create_audio_policy = create_legacy_ap;
        dev->device.destroy_audio_policy = destroy_legacy_ap;

        *device = &dev->device.common;

        return 0;
}
```

函数 create_audio_policy()中的参数 aps_ops 指针代表它是 AudioPolicyService 与外界交互的接口，具体实现代码如下所示。

```
struct audio_policy_service_ops aps_ops = {
            open_output: aps_open_output,
            open_duplicate_output : aps_open_dup_output,
            close_output: aps_close_output,
            suspend_output: aps_suspend_output,
```



```
        restore_output: aps_restore_output,
        open_input: aps_open_input,
        close_input: aps_close_input,
        set_stream_volume : aps_set_stream_volume,
        set_stream_output: aps_set_stream_output,
        set_parameters : aps_set_parameters,
        get_parameters : aps_get_parameters,
        start_tone : aps_start_tone,
        stop_tone : aps_stop_tone,
        set_voice_volume : aps_set_voice_volume,
        move_effects : aps_move_effects,
        load_hw_module : aps_load_hw_module,
        open_output_on_module : aps_open_output_on_module,
        open_input_on_module : aps_open_input_on_module,
    };
```

再看函数 create_audio_policy()，其功能是创建一个用户自定义的 policy_hal 模块的接口。假如使用的是 QCOM 的芯片，则 QCOM 有自己的一套解决方案，而原生 Android 也有自己的一套解决方案，两者的具体实现相差无几。

在函数 egacy_ap_dev_open()中存在如下代码行。

```
...
dev->device.create_audio_policy = create_legacy_ap;
...
```

由此可见调用了 create_legacy_ap()函数，此函数的对应代码如下所示。

```
static int create_legacy_ap(const struct audio_policy_device *device,
                            struct audio_policy_service_ops *aps_ops,
                            void *service,
                            struct audio_policy **ap)
{
struct legacy_audio_policy *lap;
...
lap = (struct legacy_audio_policy *)calloc(1, sizeof(*lap));
...
lap->policy.set_force_use = ap_set_force_use;
...
lap->service = service;
lap->aps_ops = aps_ops;
lap->service_client =
        new AudioPolicyCompatClient(aps_ops, service);
...
lap->apm = createAudioPolicyManager(lap->service_client);
...
*ap = &lap->policy;
...
}
```

这样，文件 AudioPolicyService.cpp 中的函数 set_force_use()调用了文件 audio_policy_hal.cpp 中的

函数 ap_set_force_use()，具体实现代码如下所示。

```
static void ap_set_force_use(struct audio_policy *pol,
                             audio_policy_force_use_t usage,
                             audio_policy_forced_cfg_t config)
{
    struct legacy_audio_policy *lap = to_lap(pol);
    lap->apm->setForceUse((AudioSystem::force_use)usage,
                             (AudioSystem::forced_config)config);
}
```

从 create_legacy_ap()函数可以得知 apm 的由来，具体实现代码如下所示。

```
lap->apm = createAudioPolicyManager(lap->service_client);
```

函数 createAudioPolicyManager()在 AudioPolicyInterface.h 接口中定义，具体实现代码如下所示。

```
extern "C" AudioPolicyInterface* createAudioPolicyManager(AudioPolicyClientInterface *clientInterface);
```

而函数 createAudioPolicyManager()由硬件厂商实现，返回其 AudioPolicyManager。而 QCOM 是在文件 AudioPolicyManagerALSA.cpp 中实现的，接下来需要根据不同的策略来切换不同的 Output 和 Input 设备以及其他一些操作。

由此可见，在 Android 的底层驱动系统中，AudioPolicyService 是一个壳子，这个壳子的核心部件是 audio_policy，真正的实现可以由厂商自己完成，当然 Android 也提供了原生驱动 AudioPolicyManagerDefault。

14.6 实现编/解码过程

为了减小传输过程中的数据流量和存储空间，传输的媒体文件必须进行压缩处理。在目前的主流移动计算平台上，支持的音频记录格式主要有 AAC 和 AMR-NB 两种。另外，部分厂商也提供了对元数据 PCM 的记录支持。

其中，AAC（Adpative Audio Coding）是在 MP3 基础上开发出来的，其主要算法早在 1997 年研发完成。与 MP3 相比，AAC 采用了修正离散余弦变换（Modified Discrete Cosiine Transform，MDCT）算法，具有更高的压缩率，能够支持最多 48 个全音域声道，最高支持 8kHz～96kHz 的采样速率，具有更高的解码效率，占用的解码资源更少。AAC 有效地解决了 MP3 的压缩率较低、音质在低码率下不够理想、仅有两个声道等问题。目前支持 AAC 的厂家主要有 Nokia、Apple、Qalcomm 和 Panasonic 等。

注意： 因为目前 OpenCORE 不支持 AAC 编码，所以本书中就不再对 AAC 的编解码进行过多的介绍。

AMR 根据带宽的不同可以分为自适应多速率宽带编码（AMR-WB，AMR WideBand）和自适应多速率窄带编码（AMR-NB，AMR NarrowBand）。其中，AMR-WB 的音频带宽为 50Hz～7000Hz，采样速率为 16kHz，而 AMR-NB 的音频带宽为 300Hz～3400Hz，采样速率为 8kHz。AMR-WB 同时被 ITU-T 和 3GPP 采用，也称 G722.2 标准。AMR-WB 抗扰度优于 AMR-NB。

在 Android 系统中，上层框架提供了 MediaRecorder 类等来支持音频内容的编码。本节将详细讲解

不同的编码和解码实现过程。

14.6.1　AMR 编码

无论是在 2G 还是 3G 通信中，AMR 都是最常用的一种音频编码格式。根据制定组织的不同，帧结构略有不同。目前业界采用的 AMR 的帧结构主要由 ETS、WMF、IETF 和 3GPP 等制定。其中，GDM&&GPRS 采用的是 ETS 制定的帧结构，WCDMA&&TD-SCDMA 采用的是 3GPP 制定的帧结构。

在 OpenCore 中，相关的编码过程保存在 external/opencore/codecs_v2/audio/gsm_amr/amr_nb/enc 目录下。支持的帧结构包括 AMR_TX_WMF、AMR_TX_IF2、AMR_TX_ETS 和 AMR_TX_IETF 等。其中，AMR_TX_WMF 表示的是无线多媒体论坛（Wireless Multimedia Forum，WMF）制定的帧结构；AMR_TX_IF2 表示的是 3GPP 制定的帧结构；AMR_TX_ETS 表示的是欧洲电信标准（European Telecommunication Standard，ETS）制定的帧结构；AMR_TX_IETF 表示的是 IETF 制定的帧结构。

编码的入口函数 AMREncode() 位于文件 opencore/codecs_v2/audio/gsm_amr/amr_nb/enc/src/amrencode.cpp 中，函数 AMREncode()首先调用 GSM EFR 编码器进行编码，然后根据指定的输出格式参数 output_format 的值，将 GSM EFR 编码器的输出转换为相应的帧结构。下面的代码是 AMR 的编码过程。

```
Word16 AMREncode(
    void *pEncState,
    void *pSidSyncState,
    enum Mode mode,
    Word16 *pEncInput,
    UWord8 *pEncOutput,
    enum Frame_Type_3GPP *p3gpp_frame_type,
    Word16 output_format
)
{
    Word16 ets_output_bfr[MAX_SERIAL_SIZE+2];
    UWord8 *ets_output_ptr;
    Word16 num_enc_bytes = -1;
    Word16 i;
    enum TXFrameType tx_frame_type;
    enum Mode usedMode = MR475;
    /*WMF 或 IF2 编码*/
    if ((output_format == AMR_TX_WMF) | (output_format == AMR_TX_IF2))
    {
        /*通话帧编码速度（20ms）*/
#ifndef CONSOLE_ENCODER_REF
        /*GSM EFR 编码器的 PV 实现*/
        GSMEncodeFrame(pEncState, mode, pEncInput, ets_output_bfr, &usedMode);
#else
        /*GSM EFR 编码器的 ETS 实现*/
        Speech_Encode_Frame(pEncState, mode, pEncInput, ets_output_bfr, &usedMode);
#endif
        /*判断帧类型*/
        sid_sync(pSidSyncState, usedMode, &tx_frame_type);
```

```
        if (tx_frame_type != TX_NO_DATA)
        {
            /*There is data to transmit*/
            *p3gpp_frame_type = (enum Frame_Type_3GPP) usedMode;
            /*为 SID 帧添加 SID 类型和模式信息*/
            if (*p3gpp_frame_type == AMR_SID)
            {
                /*Add SID type to encoder output buffer*/
                if (tx_frame_type == TX_SID_FIRST)
                {
                    ets_output_bfr[AMRSID_TXTYPE_BIT_OFFSET] &= 0x0000;
                }
                else if (tx_frame_type == TX_SID_UPDATE)
                {
                    ets_output_bfr[AMRSID_TXTYPE_BIT_OFFSET] |= 0x0001;
                }
                for (i = 0; i < NUM_AMRSID_TXMODE_BITS; i++)
                {
                    ets_output_bfr[AMRSID_TXMODE_BIT_OFFSET+i] =
                        (mode >> i) & 0x0001;
                }
            }
        }
        else
        {
            /*无数据传递*/
            *p3gpp_frame_type = (enum Frame_Type_3GPP)AMR_NO_DATA;
        }
        /*判断输出帧类型*/
        if (output_format == AMR_TX_WMF)
        {
            /*转换为 IETF 帧结构*/
            ets_to_wmf(*p3gpp_frame_type, ets_output_bfr, pEncOutput);
            /*Set up the number of encoded WMF bytes*/
            num_enc_bytes = WmfEncBytesPerFrame[(Word16) *p3gpp_frame_type];
        }
        else if (output_format == AMR_TX_IF2)
        {
            /*转换为 AMR IF2 帧结构*/
            ets_to_if2(*p3gpp_frame_type, ets_output_bfr, pEncOutput);
            num_enc_bytes = If2EncBytesPerFrame[(Word16) *p3gpp_frame_type];
        }
    }
    /*ETS 帧编码*/
    else if (output_format == AMR_TX_ETS)
    {
#ifndef CONSOLE_ENCODER_REF
        GSMEncodeFrame(pEncState, mode, pEncInput, &ets_output_bfr[1], &usedMode);
#else
        Speech_Encode_Frame(pEncState, mode, pEncInput, &ets_output_bfr[1], &usedMode);
#endif
```

```
            *p3gpp_frame_type = (enum Frame_Type_3GPP) usedMode;
            sid_sync(pSidSyncState, usedMode, &tx_frame_type);
            ets_output_bfr[0] = tx_frame_type;

            if (tx_frame_type != TX_NO_DATA)
            {
                ets_output_bfr[1+MAX_SERIAL_SIZE] = (Word16) mode;
            }
            else
            {
                ets_output_bfr[1+MAX_SERIAL_SIZE] = -1;
            }
            ets_output_ptr = (UWord8 *) & ets_output_bfr[0];
            for (i = 0; i < 2*(MAX_SERIAL_SIZE + 2); i++)
            {
                *(pEncOutput + i) = *ets_output_ptr;
                ets_output_ptr += 1;
            }
            /* Set up the number of encoded bytes */
            num_enc_bytes = 2 * (MAX_SERIAL_SIZE + 2);
        }
        /*无效的帧格式*/
        else
        {
            /*Invalid output format, set up error code*/
            num_enc_bytes = -1;
        }
        return(num_enc_bytes);
}
```

接下来需要将 GSM ETS 帧结构转换为 AMR IF2 帧结构。在文件 opencore/codecs_v2/audio/gsm_amr/amr_nb/enc/src/ets_to_if2.cpp 中，将 GSM ETS 帧结构转换为 AMR IF2 帧结构的实现代码如下所示。

```
void ets_to_if2(
    enum Frame_Type_3GPP frame_type_3gpp,
    Word16 *ets_input_ptr,
    UWord8 *if2_output_ptr)
{
    Word16   i;
    Word16   k;
    Word16   j = 0;
    Word16 *ptr_temp;
    Word16   bits_left;
    UWord8   accum;
    if (frame_type_3gpp < AMR_SID)
    {
        if2_output_ptr[j++] = (UWord8)(frame_type_3gpp) |
                              (ets_input_ptr[reorderBits[frame_type_3gpp][0]] << 4) |
                              (ets_input_ptr[reorderBits[frame_type_3gpp][1]] << 5) |
                              (ets_input_ptr[reorderBits[frame_type_3gpp][2]] << 6) |
                              (ets_input_ptr[reorderBits[frame_type_3gpp][3]] << 7);
```

```
        for (i = 4; i < numOfBits[frame_type_3gpp] - 7;)
        {
            if2_output_ptr[j]   =
                (UWord8) ets_input_ptr[reorderBits[frame_type_3gpp][i++]];
            if2_output_ptr[j] |=
                (UWord8) ets_input_ptr[reorderBits[frame_type_3gpp][i++]] << 1;
            if2_output_ptr[j] |=
                (UWord8) ets_input_ptr[reorderBits[frame_type_3gpp][i++]] << 2;
            if2_output_ptr[j] |=
                (UWord8) ets_input_ptr[reorderBits[frame_type_3gpp][i++]] << 3;
            if2_output_ptr[j] |=
                (UWord8) ets_input_ptr[reorderBits[frame_type_3gpp][i++]] << 4;
            if2_output_ptr[j] |=
                (UWord8) ets_input_ptr[reorderBits[frame_type_3gpp][i++]] << 5;
            if2_output_ptr[j] |=
                (UWord8) ets_input_ptr[reorderBits[frame_type_3gpp][i++]] << 6;
            if2_output_ptr[j++] |=
                (UWord8) ets_input_ptr[reorderBits[frame_type_3gpp][i++]] << 7;
        }
        bits_left = 4 + numOfBits[frame_type_3gpp] -
                    ((4 + numOfBits[frame_type_3gpp]) & 0xFFF8);
        if (bits_left != 0)
        {
            if2_output_ptr[j] = 0;
            for (k = 0; k < bits_left; k++)
            {
                if2_output_ptr[j] |=
                    (UWord8) ets_input_ptr[reorderBits[frame_type_3gpp][i++]] << k;
            }
        }
    }
    else
    {
        if (frame_type_3gpp != AMR_NO_DATA)
        {
            if2_output_ptr[j++] = (UWord8)(frame_type_3gpp) |
                                  (ets_input_ptr[0] << 4) | (ets_input_ptr[1] << 5) |
                                  (ets_input_ptr[2] << 6) | (ets_input_ptr[3] << 7);
            ptr_temp = &ets_input_ptr[4];
            bits_left = ((4 + numOfBits[frame_type_3gpp]) & 0xFFF8);
            for (i = (bits_left - 7) >> 3; i > 0; i--)
            {
                accum   = (UWord8) * (ptr_temp++);
                accum |= (UWord8) * (ptr_temp++) << 1;
                accum |= (UWord8) * (ptr_temp++) << 2;
                accum |= (UWord8) * (ptr_temp++) << 3;
                accum |= (UWord8) * (ptr_temp++) << 4;
                accum |= (UWord8) * (ptr_temp++) << 5;
                accum |= (UWord8) * (ptr_temp++) << 6;
                accum |= (UWord8) * (ptr_temp++) << 7;
                if2_output_ptr[j++] = accum;
```

```
            }
            bits_left = 4 + numOfBits[frame_type_3gpp] - bits_left;
            if (bits_left != 0)
            {
                if2_output_ptr[j] = 0;
                for (i = 0; i < bits_left; i++)
                {
                    if2_output_ptr[j] |= (ptr_temp[i] << i);
                }
            }
        }
        else
        {
            /* When there is no data, LSnibble of first octet */
            /* is the 3GPP frame type, MSnibble is zeroed out */
            if2_output_ptr[j++] = (UWord8)(frame_type_3gpp);
        }
    }
    return;
}
```

另外，函数 ets_to_ietf()和函数 ets_to_wmf()能够分别针对 ETS 帧结构转换为 IETF 帧结构和 WMF 帧结构，其具体实现代码分别位于文件 ets_to_if2.cpp 和 ets_to_wmf.cpp 中，读者可以查看具体实现代码，本书不再进行介绍。

14.6.2　AMR 解码

在 AMR 解码过程中，OpenCore 定义了 ETS（AMR-WB、AMR-NB）、ITU（AMR-WB）、MIME_IETF（AMR-WB）、WMF（AMR-NB）、IF2（AMR-NB）这 5 种帧结构，帧结构的定义和编码实现没有统一。在目前仅使用了 ETS、WMF、IF2 这 3 种帧结构。

AMR 的解码过程和其编码过程相反，首先需要根据输入格式参数 input_format 确定当前要解码的帧结构。如果是 IETF 或者 AMR IF2，则首先将其帧结构转换为 ETS 帧结构，然后调用函数 GSMFrameDecode()进行解码；如果是 ETS 帧结构，则直接调用函数 GSMFrameDecode()进行解码。在文件 opencore/codecs_v2/audio/gsm_amr/amr_nb/dec/src/amrdecode.cpp 中实现了 AMR 的解码过程，主要实现代码如下所示。

```
Word16 AMRDecode(
    void *state_data,
    enum Frame_Type_3GPP frame_type,
    UWord8 *speech_bits_ptr,
    Word16 *raw_pcm_buffer,
    bitstream_format input_format
)
{
    Word16 *ets_word_ptr;
    enum Mode mode = (enum Mode)MR475;
    int modeStore;
```

```
    int tempInt;
    enum RXFrameType rx_type = RX_NO_DATA;
    Word16 dec_ets_input_bfr[MAX_SERIAL_SIZE];
    Word16 i;
    Word16 byte_offset = -1;
    Speech_Decode_FrameState *decoder_state
    = (Speech_Decode_FrameState *) state_data;
    if ((input_format == MIME_IETF) | (input_format == IF2))
    {
        if (input_format == MIME_IETF)
        {
            /*转换编码*/
            wmf_to_ets(frame_type, speech_bits_ptr, dec_ets_input_bfr, &(decoder_state->decoder_amrState.
common_amr_tbls));

            /*下个框架开始的地址垂距*/
            byte_offset = WmfDecBytesPerFrame[frame_type];
        }
        else    /*必须输入 IF2 帧*/
        {
            /*转换接踵而来的 packetized 及未加工的 IF2 数据为 ETS 格式*/
            if2_to_ets(frame_type, speech_bits_ptr, dec_ets_input_bfr, &(decoder_state->decoder_amrState.
common_amr_tbls));

            /* Address offset of the start of next frame */
            byte_offset = If2DecBytesPerFrame[frame_type];
        }
        /*以 ETS 格式输入数据*/
        /*确定 AMR 编解码器方式和 AMR RX 框架类型*/
        if (frame_type <= AMR_122)
        {
            mode = (enum Mode) frame_type;
            rx_type = RX_SPEECH_GOOD;
        }
        else if (frame_type == AMR_SID)
        {
            /*在输入缓冲区前以可读方式清除*/
            modeStore = 0;
            for (i = 0; i < NUM_AMRSID_RXMODE_BITS; i++)
            {
                tempInt = dec_ets_input_bfr[AMRSID_RXMODE_BIT_OFFSET+i] << i;
                modeStore |= tempInt;
            }
            mode = (enum Mode) modeStore;

            /*得到 RX 框架类型*/
            if (dec_ets_input_bfr[AMRSID_RXTYPE_BIT_OFFSET] == 0)
            {
                rx_type = RX_SID_FIRST;
            }
```

```
                else
                {
                    rx_type = RX_SID_UPDATE;
                }
            }
            else if (frame_type < AMR_NO_DATA)
            {
                /*无效 frame_type，返回误差编码*/
                byte_offset = -1;
            }
            else
            {
                mode = decoder_state->prev_mode;
                rx_type = RX_NO_DATA;
            }
        }

        /*ETS 帧*/
        else if (input_format == ETS)
        {
            /*转换未加工的 ETS 数据*/
            ets_word_ptr = (Word16 *) speech_bits_ptr;
            /*获取 RX  帧的类型*/
            rx_type = (enum RXFrameType) * ets_word_ptr;
            ets_word_ptr++;
            for (i = 0; i < MAX_SERIAL_SIZE; i++)
            {
                dec_ets_input_bfr[i] = *ets_word_ptr;
                ets_word_ptr++;
            }
            /*得到编解码器方式*/
            if (rx_type != RX_NO_DATA)
            {
                /* Get mode from input bitstream */
                mode = (enum Mode) * ets_word_ptr;
            }
            else
            {
                /*如果没有收到数据*/
                mode = decoder_state->prev_mode;
            }
            byte_offset = 2 * (MAX_SERIAL_SIZE + 2);
        }
        else
        {/
            byte_offset = -1;
        }
        if (byte_offset != -1)
        {
#ifndef CONSOLE_DECODER_REF
```

```
            GSMFrameDecode(decoder_state, mode, dec_ets_input_bfr, rx_type,
                                raw_pcm_buffer);
#else
            Speech_Decode_Frame(decoder_state, mode, dec_ets_input_bfr, rx_type,
                                    raw_pcm_buffer);
#endif
            decoder_state->prev_mode = mode;
      }
      return (byte_offset);
}
```

在文件 codecs_v2/audio/gsm_amr/amr_nb/dec/src/if2_to_ets.cpp 中定义函数 if2_to_ets()，用于将 AMR IF2 帧结构转换为 ETS 帧结构。函数 if2_to_ets()的主要实现代码如下所示。

```
void if2_to_ets(
      enum Frame_Type_3GPP frame_type_3gpp,
      UWord8 *if2_input_ptr,
      Word16 *ets_output_ptr,
      CommonAmrTbls* common_amr_tbls)
{

      Word16 i;
      Word16 j;
      Word16 x = 0;
      const Word16* numCompressedBytes_ptr = common_amr_tbls->numCompressedBytes_ptr;
      const Word16* numOfBits_ptr = common_amr_tbls->numOfBits_ptr;
      const Word16* const* reorderBits_ptr = common_amr_tbls->reorderBits_ptr;
      if (frame_type_3gpp < AMR_SID)
      {
            for (j = 4; j < 8; j++)
            {
                  ets_output_ptr[reorderBits_ptr[frame_type_3gpp][x++]] =
                        (if2_input_ptr[0] >> j) & 0x01;
            }
            for (i = 1; i < numCompressedBytes_ptr[frame_type_3gpp]; i++)
            {
                  for (j = 0; j < 8; j++)
                  {
                        if (x >= numOfBits_ptr[frame_type_3gpp])
                        {
                              break;
                        }
                        ets_output_ptr[reorderBits_ptr[frame_type_3gpp][x++]] =
                              (if2_input_ptr[i] >> j) & 0x01;
                  }
            }
      }
      else
      {
            for (j = 4; j < 8; j++)
            {
```

```
                ets_output_ptr[x++] =
                    (if2_input_ptr[0] >> j) & 0x01;
            }
            for (i = 1; i < numCompressedBytes_ptr[frame_type_3gpp]; i++)
            {
                for (j = 0; j < 8; j++)
                {
                    ets_output_ptr[x++] =
                        (if2_input_ptr[i] >> j) & 0x01;
                }
            }
        }
    return;
}
```

在目前的解码实现上，仅支持 AMR IF2、ETS 帧结构，并不支持 IETF 帧结构。另外，函数 wmf_to_ets()的具体实现位于文件 amrdecode.cpp 中，本书将不再介绍此函数，读者可参阅其具体实现代码。

14.6.3　解码 MP3

MP3（Moving Picture Experts Group Audio Layer III）是目前最流行的音频编码格式，MP3 的解码需要经过同步及检错、哈夫曼解码、逆量化、立体声解码、反锯齿、IMDCT 和子带合成等运算，其中，IMDCT 过程的运算量占到了整个解码运算总量的 19%。

在文件 opencore/codecs_v2/omx/omx_mp3/src/mp3_dec.cpp 中实现了对 MP3 文件的解码，主要实现代码如下所示。

```
int Mp3Decoder::Mp3DecodeAudio(OMX_S16* aOutBuff,
                               OMX_U32* aOutputLength, OMX_U8** aInputBuf,
                               OMX_U32* aInBufSize, OMX_S32* aIsFirstBuffer,
                               OMX_AUDIO_PARAM_PCMMODETYPE* aAudioPcmParam,
                               OMX_AUDIO_PARAM_MP3TYPE* aAudioMp3Param,
                               OMX_BOOL aMarkerFlag,
                               OMX_BOOL* aResizeFlag)
{
    int32 Status = MP3DEC_SUCCESS;
    *aResizeFlag = OMX_FALSE;
    if (iInitFlag == 0)
    {
        if (*aIsFirstBuffer != 0)
        {
            e_equalization EqualizType = iMP3DecExt->equalizerType;
            iMP3DecExt->inputBufferCurrentLength = 0;
            iInputUsedLength = 0;
            iAudioMp3Decoder->StartL(iMP3DecExt, false, false, false, EqualizType);
        }
        iInitFlag = 1;
    }
    iMP3DecExt->pInputBuffer = *aInputBuf + iInputUsedLength;
```

```
iMP3DecExt->pOutputBuffer = &aOutBuff[0];
iMP3DecExt->inputBufferCurrentLength = *aInBufSize;
iMP3DecExt->inputBufferUsedLength = 0;
if (OMX_FALSE == aMarkerFlag)
{
    //如果没有标志位，则检测帧的边界
    Status = iAudioMp3Decoder->SeekMp3Synchronization(iMP3DecExt);
    if (1 == Status)
    {
        if (0 == iMP3DecExt->inputBufferCurrentLength)
        {
            *aInBufSize -= iMP3DecExt->inputBufferMaxLength;
            iInputUsedLength += iMP3DecExt->inputBufferMaxLength;
            iMP3DecExt->inputBufferUsedLength += iMP3DecExt->inputBufferMaxLength;;
            return MP3DEC_SUCCESS;
        }
        else
        {
            *aInputBuf += iInputUsedLength;
            iMP3DecExt->inputBufferUsedLength = 0;
            iInputUsedLength = 0;
            return MP3DEC_INCOMPLETE_FRAME;
        }
    }
}
Status = iAudioMp3Decoder->ExecuteL(iMP3DecExt);
if (MP3DEC_SUCCESS == Status)
{
    *aInBufSize -= iMP3DecExt->inputBufferUsedLength;
    if (0 == *aInBufSize)
    {
        iInputUsedLength = 0;
    }
    else
    {
        iInputUsedLength += iMP3DecExt->inputBufferUsedLength;
    }
    *aOutputLength = iMP3DecExt->outputFrameSize * iMP3DecExt->num_channels;
    if (0 == *aIsFirstBuffer)
    {
        (*aIsFirstBuffer)++;
        aAudioPcmParam->nSamplingRate = iMP3DecExt->samplingRate;
        aAudioPcmParam->nChannels = iMP3DecExt->num_channels;
        *aResizeFlag = OMX_TRUE;
    }
    return Status;
}
else if (Status == MP3DEC_INVALID_FRAME)
{
    *aInBufSize = 0;
```

```
            iInputUsedLength = 0;
        }
        else if (Status == MP3DEC_INCOMPLETE_FRAME)
        {
            *aInputBuf += iInputUsedLength;
            iMP3DecExt->inputBufferUsedLength = 0;
            iInputUsedLength = 0;
        }
        else
        {
            *aInputBuf += iInputUsedLength;
            iInputUsedLength = 0;
        }
        return Status;
}
```

第 15 章　视频系统架构详解

在当前的智能手机系统应用中，多媒体视频应用比较常见，用户通常在手机中播放各种各样的视频文件，也常用手机在线观看视频。在 Android 系统中，为开发人员提供了功能强大的视频系统框架。本章将详细讲解 Android 5.0 系统中各个视频框架的架构知识，为读者学习本书后面的知识打下基础。

15.1　视频输出系统

在 Android 系统中，视频输出系统对应的是 Overlay 子系统，此系统是 Android 的一个可选系统，用于加速显示输出视频数据。视频输出系统的硬件通常叠加在主显示区之上额外的叠加显示区。这个额外的叠加显示区和主显示区使用独立的显示内存。在通常情况下，主显示区用于输出图形系统，通常是 RGB 颜色空间。额外显示区用于输出视频，通常是 YUV 颜色空间。主显示区和叠加显示区通过 Blending（硬件混淆）自动显示在屏幕上。在软件部分无须关心叠加的实现过程，但是可以控制叠加的层次顺序和叠加层的大小等内容。本节将详细讲解 Overlay 视频输出系统的架构知识。

15.1.1　基本层次结构

在 Android 系统中，Overlay 系统的基本层次结构如图 15-1 所示。

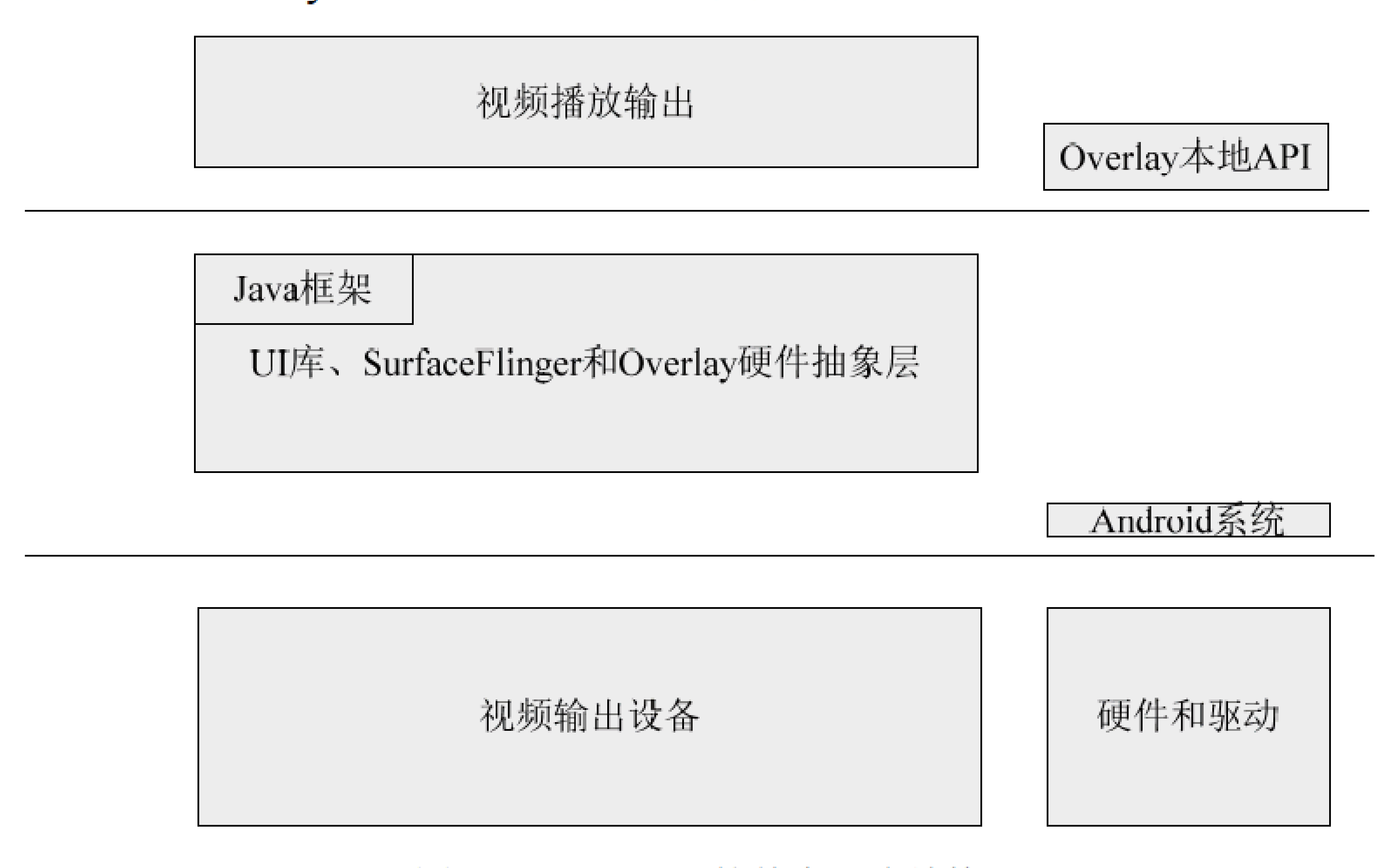


图 15-1　Overlay 的基本层次结构

Android 中的 Overlay 系统没有 Java 部分，只包含了视频输出的驱动程序、硬件抽象层和本地框架等。Overlay 系统的结构如图 15-2 所示。

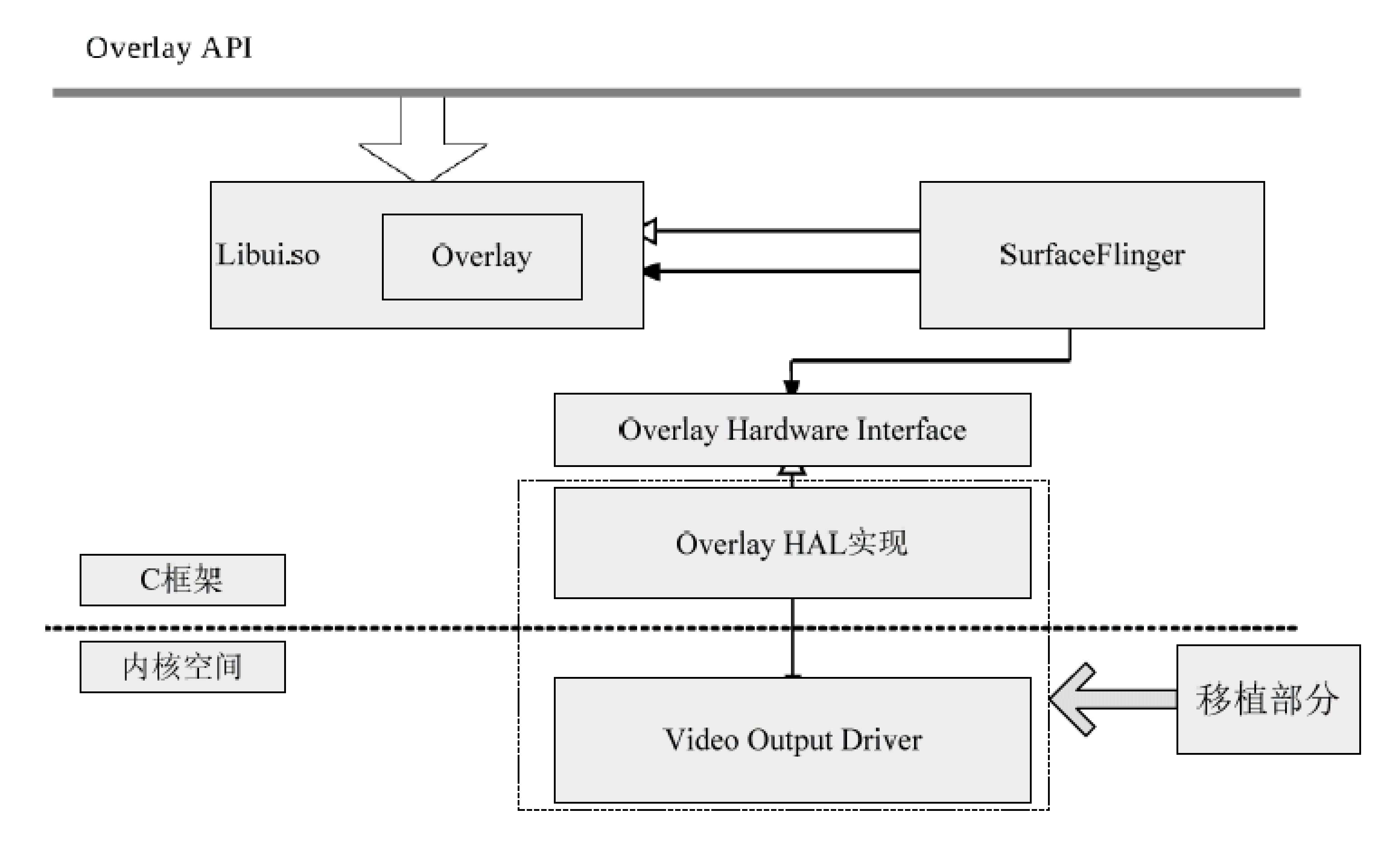


图 15-2　Overlay 系统结构

在图 15-2 所示的系统结构中，各个构成部分的具体说明如下所示。

（1）Overlay 驱动程序：通常是基于 FrameBuffer 或 V4L2 的驱动程序。在此文件中主要定义了两个 struct，分别是 data device 和 control device，这两个结构体分别针对 data device 和 control device 的函数 open()和 close()。这两个函数是注册到 device_module 中的函数。

（2）Overlay 硬件抽象层：代码路径为 hardware/qcom/display/liboverlay/overlay.h。

Overlay 硬件抽象层是一个 Android 中标准的硬件模块，其接口只有一个头文件。

（3）Overlay 服务部分：代码路径为 frameworks/native/services/surfaceflinger/。

由此可见，Overlay 系统的服务部分包含在 SurfaceFlinger 中，此层次的内容比较简单，主要功能是通过类 LayerBuffer 实现的。首先要明确的是 SurfaceFlinger 只是负责控制 merge Surface，例如，计算出两个 Surface 重叠的区域，至于 Surface 需要显示的内容，则通过 Skia、OpenGL 和 Pixflinger 来计算。所以在介绍 SurfaceFlinger 之前先忽略里面存储的内容究竟是什么，先明确它对 merge 的一系列控制的过程，然后再结合 2D、3D 引擎来看其处理过程。

（4）本地框架代码。

在 Overlay 系统中，本地框架的头文件路径为 frameworks/native/include/ui。

源代码路径为 frameworks/native/libs/ui。

Overlay 系统只是整个框架的一部分，主要功能是通过类 Ioverlay 和 Overlay 实现的，源代码被编译成 libui.so，所提供的 API 主要在视频输出和照相机取景模块中使用。

15.1.2　硬件抽象层架构

Overlay 系统的硬件抽象层是一个硬件模块，下面将简要介绍 Overlay 系统的硬件抽象层的基本知识，为后面的知识做好铺垫。

1. Overlay 系统硬件抽象层的接口

在 Android 系统中，通过文件 hardware/qcom/display/liboverlay/overlay.h 定义 Overlay 系统硬件抽象层的接口。

在文件 overlay.h 中，主要定义了 data device 和 control device 两个 struct，并提供针对 data device

和 control device 的函数 open()和 close()。文件 overlay.h 的代码结构如图 15-3 所示。

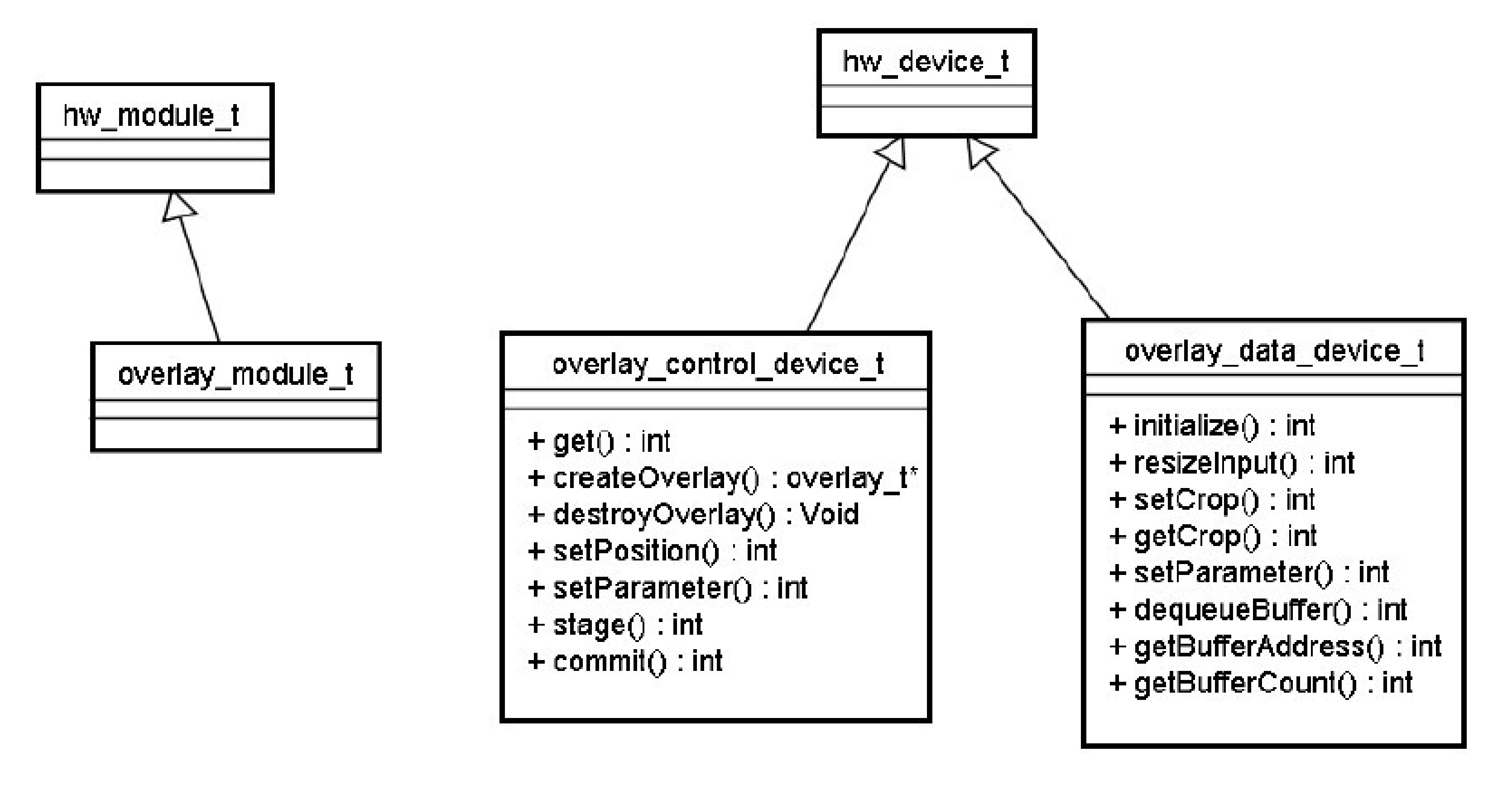


图 15-3　文件 overlay.h 的代码结构

（1）定义 Overlay 控制设备和 Overlay 数据设备，其名称被定义为如下两个字符串。

```
#define OVERLAY_HARDWARE_CONTROL "control"
#define OVERLAY_HARDWARE_DATA "data"
```

（2）定义一个枚举 enum，定义了所有支援的 Format，FrameBuffer 会根据 Format 和 width、height 来决定 Buffer（FrameBuffer 中用来显示的 Buffer）的大小。定义 enum 的代码如下所示。

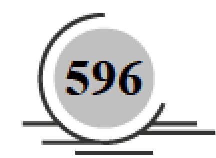
```
enum {
    OVERLAY_FORMAT_RGBA_8888 = HAL_PIXEL_FORMAT_RGBA_8888,
    OVERLAY_FORMAT_RGB_565 = HAL_PIXEL_FORMAT_RGB_565,
    OVERLAY_FORMAT_BGRA_8888 = HAL_PIXEL_FORMAT_BGRA_8888,
    OVERLAY_FORMAT_YCbCr_422_SP = HAL_PIXEL_FORMAT_YCbCr_422_SP,
    OVERLAY_FORMAT_YCbCr_420_SP = HAL_PIXEL_FORMAT_YCbCr_420_SP,
    OVERLAY_FORMAT_YCrCb_420_SP = HAL_PIXEL_FORMAT_YCrCb_420_SP,
    OVERLAY_FORMAT_YCbYCr_422_I = HAL_PIXEL_FORMAT_YCbCr_422_I,
    OVERLAY_FORMAT_YCbYCr_420_I = HAL_PIXEL_FORMAT_YCbCr_420_I,
    OVERLAY_FORMAT_CbYCrY_422_I = HAL_PIXEL_FORMAT_CbYCrY_422_I,
    OVERLAY_FORMAT_CbYCrY_420_I = HAL_PIXEL_FORMAT_CbYCrY_420_I,
    OVERLAY_FORMAT_DEFAULT = 99

};
```

（3）定义和 Overlay 系统相关结构体。

在文件 overlay.h 中和 Overlay 系统相关结构体是 overlay_t 和 overlay_handle_t，主要代码如下所示。

```
typedef struct overlay_t {
    uint32_t w;                     //宽
    uint32_t h;                     //高
    int32_t format;                 //颜色格式
    uint32_t w_stride;              //一行的内容
    uint32_t h_stride;              //一列的内容
    uint32_t reserved[3];
    /* returns a reference to this overlay's handle (the caller doesn't
```



```
     * take ownership) */
    overlay_handle_t (*getHandleRef)(struct overlay_t* overlay);
    uint32_t reserved_procs[7];
} overlay_t;
```

结构体 overlay_handle_t 是在内部使用的结构体，用于保存 Overlay 硬件设备的句柄。在使用的过程中，需要从 overlay_t 获取 overlay_handle_t。其中，上一层的使用只实现结构体 overlay_handle_t 指针的传递，具体的操作是在 Overlay 的硬件抽象层中完成的。

（4）定义结构体 overlay_control_device_t，此结构体定义了一个 control device，里面的成员除了 common 都是函数，这些函数就是需要实现的，在实现时会基于这个结构体扩展出一个关于 control device 的 context 的结构体，context 结构体内部会扩充一些信息并且包含 control device。每一个 device 中必须有 Common，而且必须放到第一位，目的只是为了使 overlay_control_device_t 和 hw_device_t 相匹配。overlay_control_device_t 的定义代码如下所示。

```
struct overlay_control_device_t {
    struct hw_device_t common;
    int (*get)(struct overlay_control_device_t *dev, int name);
    //建立设备
    overlay_t* (*createOverlay)(struct overlay_control_device_t *dev,
            uint32_t w, uint32_t h, int32_t format);
    //释放资源，分配的 handle 和 control device 的内存
    void (*destroyOverlay)(struct overlay_control_device_t *dev,
            overlay_t* overlay);
    //设置 overlay 的显示范围。（如果是 camera 的 preview，那么 h、w 要和 preview h、w 一致）
    int (*setPosition)(struct overlay_control_device_t *dev,
            overlay_t* overlay,
            int x, int y, uint32_t w, uint32_t h);
    //获取 overlay 的显示范围
    int (*getPosition)(struct overlay_control_device_t *dev,
            overlay_t* overlay,
            int* x, int* y, uint32_t* w, uint32_t* h);
    int (*setParameter)(struct overlay_control_device_t *dev,
            overlay_t* overlay, int param, int value);
    int (*stage)(struct overlay_control_device_t *dev, overlay_t* overlay);
    int (*commit)(struct overlay_control_device_t *dev, overlay_t* overlay);
};
```

（5）定义结构体 overlay_data_device_t，此结构体和 overlay_control_device_t 类似。在具体使用上，overlay_control_device_t 负责初始化、销毁和控制类的操作，overlay_data_device_t 用于显示内存输出的数据操作。结构体 overlay_data_device_t 的定义代码如下所示。

```
struct overlay_data_device_t {
    struct hw_device_t common;
    //通过参数 handle 来初始化 data device
    int (*initialize)(struct overlay_data_device_t *dev,
            overlay_handle_t handle);
    //重新配置显示参数 w 和 h。使这两个参数生效，此处需要先 close，然后重新 open
    int (*resizeInput)(struct overlay_data_device_t *dev,
            uint32_t w, uint32_t h);
```

```
    //下面分别设置显示的区域和获取显示的区域，当播放时，需要其坐标和宽高来定义如何显示这些数据
    int (*setCrop)(struct overlay_data_device_t *dev,
            uint32_t x, uint32_t y, uint32_t w, uint32_t h) ;
    int (*getCrop)(struct overlay_data_device_t *dev,
        uint32_t* x, uint32_t* y, uint32_t* w, uint32_t* h) ;
    int (*setParameter)(struct overlay_data_device_t *dev,
            int param, int value);
    int (*dequeueBuffer)(struct overlay_data_device_t *dev,
                overlay_buffer_t *buf);
    int (*queueBuffer)(struct overlay_data_device_t *dev,
            overlay_buffer_t buffer);
    void* (*getBufferAddress)(struct overlay_data_device_t *dev,
            overlay_buffer_t buffer);
    int (*getBufferCount)(struct overlay_data_device_t *dev);
    int (*setFd)(struct overlay_data_device_t *dev, int fd);
};
```

2．实现 Overlay 系统的硬件抽象层

在 Android 系统中，提供了一个 Overlay 硬件抽象层的框架实现，其中有完整的实现代码，可以将其作为使用 Overlay 硬件抽象层的方法，但是没有使用具体硬件，所以不会有实际的实现效果。上述框架实现的源码目录为 hardware/libhardware/modules/overlay/。

在上述目录中，主要包含了文件 Android.mk 和 overlay.cpp，其中，文件 Android.mk 的主要代码如下所示。

```
LOCAL_PATH := $(call my-dir)

# HAL module implemenation, not prelinked and stored in
# hw/<OVERLAY_HARDWARE_MODULE_ID>.<ro.product.board>.so
include $(CLEAR_VARS)
LOCAL_PRELINK_MODULE := false
LOCAL_MODULE_PATH := $(TARGET_OUT_SHARED_LIBRARIES)/hw
LOCAL_SHARED_LIBRARIES := liblog
LOCAL_SRC_FILES := overlay.cpp
LOCAL_MODULE := overlay.trout
include $(BUILD_SHARED_LIBRARY)
```

Overlay 库是一个 C 语言库，没有被其他库所链接，使用时是被动打开的，所以必须被放置在目标文件系统的 system/lib/hw 目录中。

文件 overlay.cpp 的主要代码如下所示。

```
//此结构体用于扩充 overlay_control_device_t 结构体
struct overlay_control_context_t {
    struct overlay_control_device_t device;
    /* our private state goes below here */
};
//此结构体用于扩充 overlay_data_device_t 结构体
struct overlay_data_context_t {
    struct overlay_data_device_t device;
    /* our private state goes below here */
```

```
};

//定义打开函数
static int overlay_device_open(const struct hw_module_t* module, const char* name,
            struct hw_device_t** device);

static struct hw_module_methods_t overlay_module_methods = {
      open: overlay_device_open
};
struct overlay_module_t HAL_MODULE_INFO_SYM = {
      common: {
            tag: HARDWARE_MODULE_TAG,
            version_major: 1,
            version_minor: 0,
            id: OVERLAY_HARDWARE_MODULE_ID,
            name: "Sample Overlay module",
            author: "The Android Open Source Project",
            methods: &overlay_module_methods,
      }
static int overlay_device_open(const struct hw_module_t* module, const char* name,
            struct hw_device_t** device)
{
      int status = -EINVAL;
      if (!strcmp(name, OVERLAY_HARDWARE_CONTROL)) {                //Overlay 的控制设备
            struct overlay_control_context_t *dev;
            dev = (overlay_control_context_t*)malloc(sizeof(*dev));

            /* initialize our state here */
            memset(dev, 0, sizeof(*dev));                           //初始化结构体

            /* initialize the procs */
            dev->device.common.tag = HARDWARE_DEVICE_TAG;
            dev->device.common.version = 0;
            dev->device.common.module = const_cast<hw_module_t*>(module);
            dev->device.common.close = overlay_control_close;

            dev->device.get = overlay_get;
            dev->device.createOverlay = overlay_createOverlay;
            dev->device.destroyOverlay = overlay_destroyOverlay;
            dev->device.setPosition = overlay_setPosition;
            dev->device.getPosition = overlay_getPosition;
            dev->device.setParameter = overlay_setParameter;

            *device = &dev->device.common;
            status = 0;
      } else if (!strcmp(name, OVERLAY_HARDWARE_DATA)) {       //Overlay 的数据设备
            struct overlay_data_context_t *dev;
            dev = (overlay_data_context_t*)malloc(sizeof(*dev));

            /* initialize our state here */
```

```
            memset(dev, 0, sizeof(*dev));                              //初始化结构体

            /* initialize the procs */
            dev->device.common.tag = HARDWARE_DEVICE_TAG;
            dev->device.common.version = 0;
            dev->device.common.module = const_cast<hw_module_t*>(module);
            dev->device.common.close = overlay_data_close;

            dev->device.initialize = overlay_initialize;
            dev->device.dequeueBuffer = overlay_dequeueBuffer;
            dev->device.queueBuffer = overlay_queueBuffer;
            dev->device.getBufferAddress = overlay_getBufferAddress;

            *device = &dev->device.common;
            status = 0;
    }
    return status;
}
```

在实现 Overlay 系统的硬件抽象层时，具体实现方法取决于硬件和驱动程序，根据设备需要进行处理。具体来说分为如下两种情况。

（1）FrameBuffer 驱动程序方式

在此方式下，需要先实现函数 getBufferAddress()，返回通过 mmap 获得 FrameBuffer 的指针即可。如果没有双缓冲的问题，不需要真正实现函数 dequeueBuffer()和 queueBuffer()。上述函数的实现文件是 overlay.cpp，此文件被保存在目录 Hardware/qcom/display/liboverlay/overlay.cpp 中。

函数 getBufferAddress()用于返回 FrameBuffer 内部显示的内存，通过 mmap 获取内存地址。函数代码如下所示。

```
void* Overlay::getBufferAddress(overlay_buffer_t buffer)
{
    if (mStatus != NO_ERROR) return NULL;
    return mOverlayData->getBufferAddress(mOverlayData, buffer);
}
```

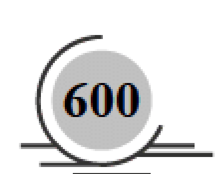
函数 dequeueBuffer()和 queueBuffer()的实现代码如下所示。

```
status_t Overlay::dequeueBuffer(overlay_buffer_t* buffer)
{
    if (mStatus != NO_ERROR) return mStatus;
    return    mOverlayData->dequeueBuffer(mOverlayData, buffer);
}

status_t Overlay::queueBuffer(overlay_buffer_t buffer)
{
    if (mStatus != NO_ERROR) return mStatus;
    return mOverlayData->queueBuffer(mOverlayData, buffer);
}
```

（2）Video for Linux 2 方式

如果使用 Video for Linux 2 的输出驱动，函数 dequeueBuffer()和 queueBuffer()调用驱动时，主要 ioctl



是一致的，即分别调用 VIDIOC_QBUF 和 VIDIOC_DQBUF 即可直接实现。至于其他的初始化工作，可以在 initialize 中进行处理。因为存在视频数据队列，所以此处处理的内容比一般的帧缓冲区要复杂，但是可以实现更高的性能。

由此可见，在某一个硬件系统中，Overlay 的硬件层和 Overlay 系统的调用者都是特定实现的，所以只需匹配上下层代码即可实现，并不需要一一满足每一个要求，各个接口可以根据具体情况灵活使用。

3．实现接口

在 Android 系统中，Overlay 系统提供了接口 overlay，此接口用于叠加在主显示层上面的另外一个显示层。此叠加的显示层经常作为视频的输出或相机取景器的预览界面来使用。文件 Overlay.h 的主要内部实现类是 Overlay 和 overlayRef。OverlayRef 需要和 surface 配合使用，通过 Isurface 可以创建出 OverlayRef。RefBase 的主要代码如下所示。

```
class Overlay : public virtual RefBase
{
public:
    Overlay(const sp<OverlayRef>& overlayRef);
    void destroy();
    //获取 overlay handle，可以根据自己的需要扩展，扩展之后有很多数据
    overlay_handle_t getHandleRef() const;
    //获取 framebuffer，用于显示内存地址
    status_t dequeueBuffer(overlay_buffer_t* buffer);
    status_t queueBuffer(overlay_buffer_t buffer);
    status_t resizeInput(uint32_t width, uint32_t height);
    status_t setCrop(uint32_t x, uint32_t y, uint32_t w, uint32_t h) ;
    status_t getCrop(uint32_t* x, uint32_t* y, uint32_t* w, uint32_t* h) ;
    status_t setParameter(int param, int value);
    void* getBufferAddress(overlay_buffer_t buffer);

    /*获取属性的信息*/
    uint32_t getWidth() const;
    uint32_t getHeight() const;
    int32_t getFormat() const;
    int32_t getWidthStride() const;
    int32_t getHeightStride() const;
    int32_t getBufferCount() const;
    status_t getStatus() const;

private:
    virtual ~Overlay();

    sp<OverlayRef> mOverlayRef;
    overlay_data_device_t *mOverlayData;
    status_t mStatus;
};

Overlay(const sp<OverlayRef>& overlayRef);
```

在上述代码中，通过 surface 来控制 Overlay，也可以在不使用 Overlay 的情况下统一进行管理。此处通过 overlayRef 来创建 Overlay，一旦获取了 Overlay 即可通过这个 Overlay 来获取到用来显示的

Address 地址，向 Address 中写入数据后即可显示图像。

15.2　MediaPlayer 架构详解

在 Android 系统中，MediaPlayer 既可以播放音频，也可以播放视频。本节将详细讲解 MediaPlayer 系统的基本架构知识，为读者学习本书后面的知识打下基础。

15.2.1　MediaPlayer 架构图解

在 Android 原生系统中，是由 mediaplayerservice 来控制媒体播放器的。在 MediaPlayerService 中创建了 MediaPlayer，在文件 mediaplayer.java 中，native 方法通过 JNI 调用 android_media_mediaplayer.cpp 中的方法，接着往下调用 mediaplayer.cpp 中的方法，mediaplayer 通过 IPC 机制调用 MediaPlayerService 中的方法。MediaPlayerService 通过对文件格式的判断来选择不同的播放器播放音乐，当是 MIDI 格式时会选择 Sonivox 来播放。当系统的配置文件中允许 OGG 格式由 vorbris 来播放时，则用 vorbris，否则用 StageFright 来播放。其余的格式由配置文件选择是否由 StageFright 来播放，是则由 StageFright 播放，不是则由 OpenCore 的 PvPlayer 来播放。

StageFright 由 AweSongPlayer 来控制，调用 setDatasource()方法来加载音频文件，根据音频文件的头字段不同来选择不同的解析器，这个解析器会进行 A/V 分离操作，分离出 audioTrack 和 videoTrack。接着会根据 audioTrack 的 mineType 类型来选择不同的编码器编码，此时由 audioSource 进行解码。audioSource 是对 omxCodec 的封装，而 audioPlayer 则是用来控制 audioSource 和 audioTrack 的。Audioplayer 通过调用 fillBuffer()方法将解码后的数据写进 data 中，最终将解码的数据流传给 audioTrack，由 audioTrack 交给 audioFlinger，audioTrack 通过调用 creataudioTrack()得到 audioFlinger 返回的 iaudioTrack，将数据流写进 iaudioTrack 的共享 Buffer 中，然后 audioFlinger 读出缓存中的数据，并交给 playbackTread 进行混音处理，或者直接输出给缓存并最终将数据交给 audioOutputStream 处理。

上述流程的具体架构如图 15-4 所示。

在 Android 系统中，MediaPlayer 在底层是基于 OpenCore（PacketVideo）的库实现的。为了构建一个 MediaPlayer 程序，在上层还包含了进程间通信等内容，这种进程间通信的基础是 Android 基本库中的 Binder 机制。

以 Android 5.0 系统为例，MediaPlayer 系统的代码主要在以下目录中实现。

（1）Java 程序的路径为 packages/apps/Music/src/com/android/music/。

Java 类的路径为 frameworks/base/media/java/android/media/MediaPlayer.java。

（2）Java 本地调用部分（JNI）：frameworks/base/media/jni/android_media_MediaPlayer.cpp。

这部分内容编译成 libmedia_jni.so，主要的头文件在目录 frameworks/base/include/media/中实现。

多媒体底层库在目录 frameworks/base/media/libmedia/中实现。

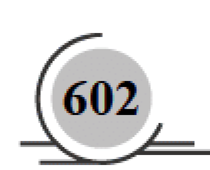
这部分的内容被编译成库 libmedia.so。

（3）多媒体服务部分：frameworks/av/media/libmediaplayerservice。

核心实现文件为 mediaplayerservice.h 和 mediaplayerservice.cpp，这部分内容被编译成库 libmediaplayerservice.so。

（4）基于 OpenCore 的多媒体播放器部分：external/opencore/。

这部分内容被编译成库 libopencoreplayer.so。



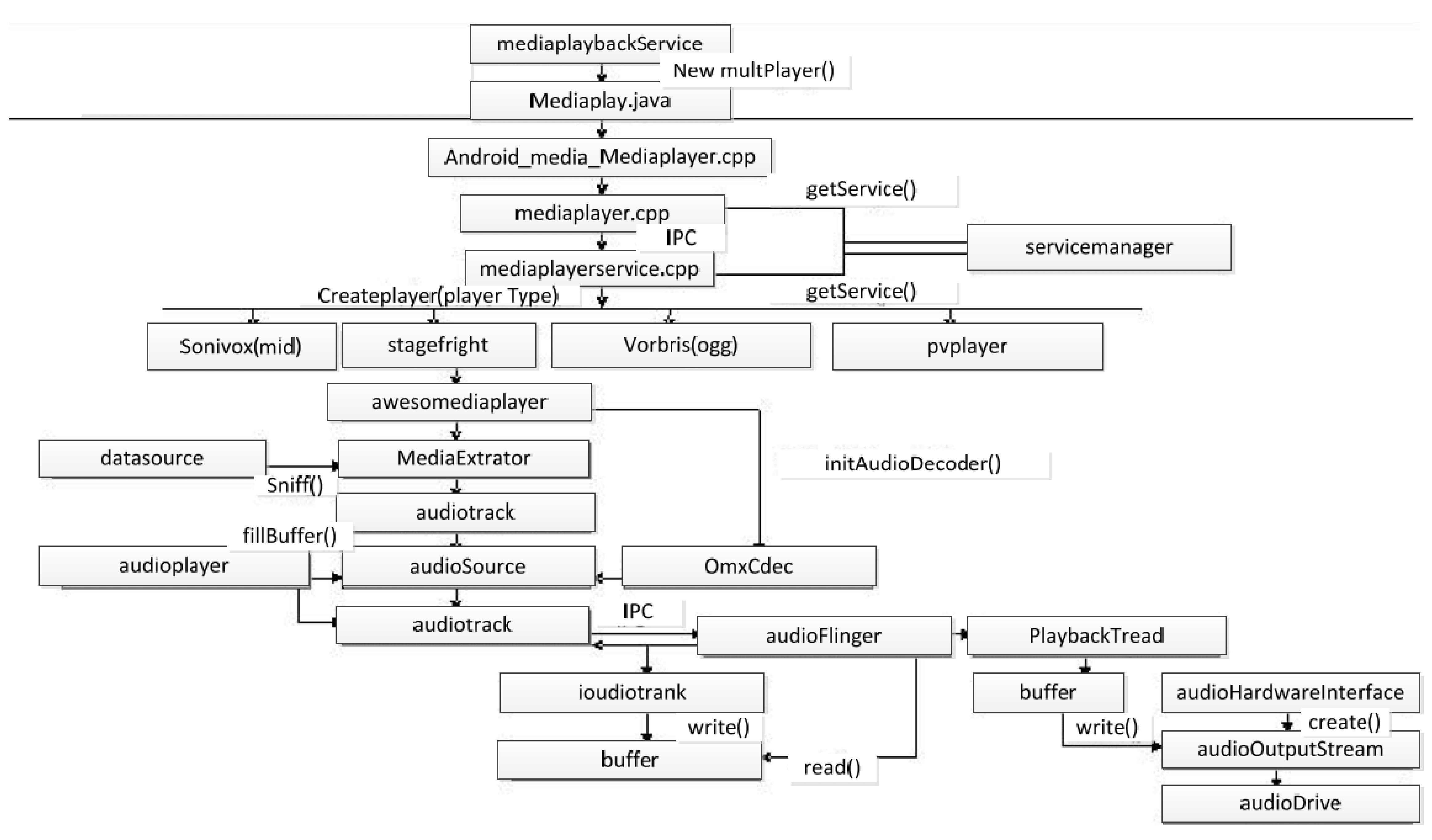


图 15-4　MediaPlayer 架构图

在上述各个部分中，libopencoreplayer.so 是最主要的实现部分，而其他的库基本上都是在其上建立的封装和为进程间通信建立的机制。

15.2.2　MediaPlayer 的接口与架构

在 Android 系统中，各个 MediaPlayer 库的结构比较复杂，具体如图 15-5 所示。

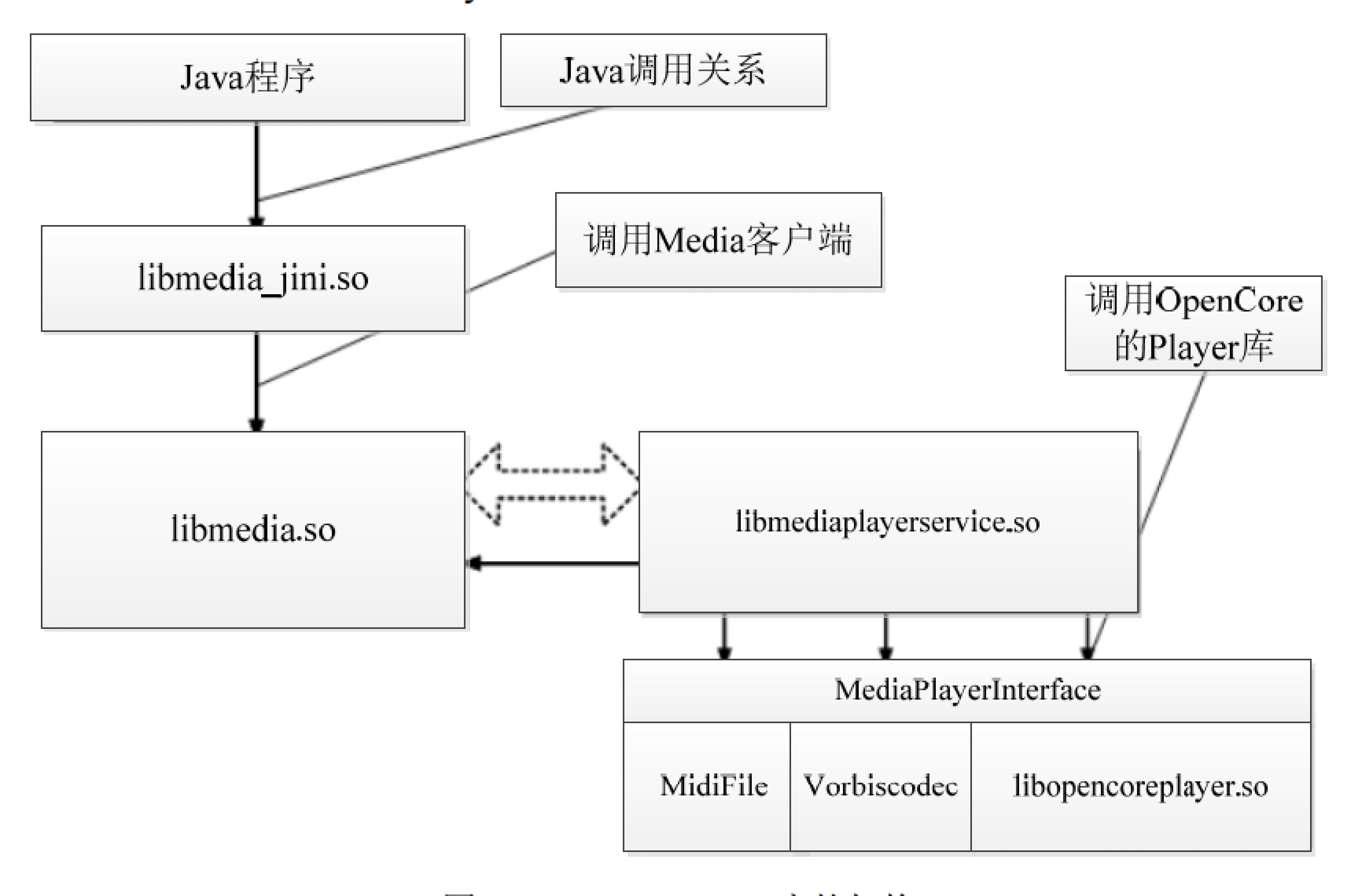


图 15-5　MediaPlayer 库的架构

- ☑ libmedia.so 库：位于核心的位置，对上层提供的接口主要是 MediaPlayer 类。libmedia_jni.so 类不但通过调用 MediaPlayer 类提供了对 Java 层的接口，而且实现了 android.media.MediaPlayer 类。
- ☑ libmediaplayerservice.so 库：是 Media 的服务器，通过继承 libmedia.so 的类实现服务器的功能，而 libmedia.so 中的另外一部分内容则通过进程间通信和 libmediaplayerservice.so 进行通信。libmediaplayerservice.so 的真正功能通过调用 OpenCore Player 来完成。

MediaPlayer 部分的头文件在 frameworks/av/include/media 目录中实现，主要包含如下头文件。

- ☑ IMediaPlayerClient.h
- ☑ mediaplayer.h
- ☑ IMediaPlayer.h
- ☑ IMediaPlayerService.h
- ☑ MediaPlayerInterface.h

其中，头文件 mediaplayer.h 提供了对上层的接口，而其他的几个头文件都是提供一些接口类（即包含了纯虚函数的类），这些接口类必须被实现类继承才能够使用。

整个 MediaPlayer 库之间的调用关系如图 15-6 所示。

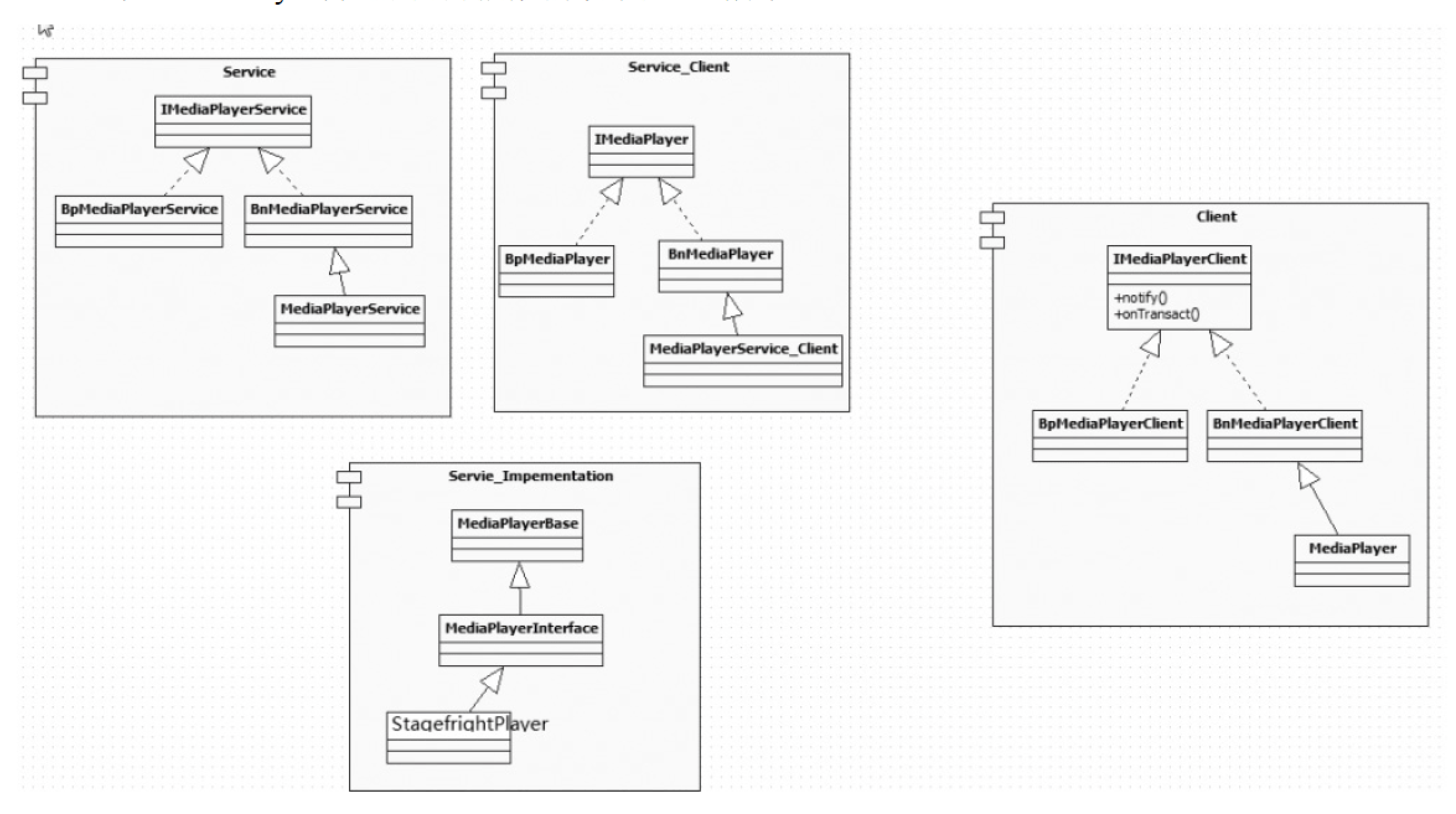
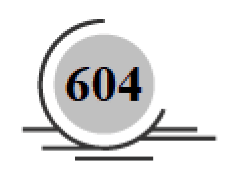


图 15-6　MediaPlayer 库之间的调用关系

在运行 MediaPlayer 时，可以将整个过程分为 Client 和 Server 两个部分，分别在两个进程中运行，之间使用 Binder 机制实现 IPC 通信。从框架结构上来看，IMediaPlayerService.h、IMediaPlayerClient.h 和 MediaPlayer.h 这 3 个文件类定义了 MediaPlayer 的接口和架构，文件 MediaPlayerService.cpp 和 mediaplayer.cpp 用于实现 MediaPlayer 架构，MediaPlayer 的具体功能在 PVPlayer（libopencoreplayer.so 库）中实现。

（1）头文件 IMediaPlayerClient.h

头文件 IMediaPlayerClient.h 的功能是描述一个 MediaPlayer 客户端的接口，具体代码如下所示。



```
namespace android {

class IMediaPlayerClient: public IInterface
{
public:
    DECLARE_META_INTERFACE(MediaPlayerClient);

    virtual void notify(int msg, int ext1, int ext2, const Parcel *obj) = 0;
};

// ----------------------------------------------------------------------------

class BnMediaPlayerClient: public BnInterface<IMediaPlayerClient>
{
public:
    virtual status_t    onTransact( uint32_t code,
                                    const Parcel& data,
                                    Parcel* reply,
                                    uint32_t flags = 0);
};

};
```

在上述代码中，类 IMediaPlayerClient 继承于 IInterface 接口，并定义了一个 MediaPlayer 客户端的接口。类 BnMediaPlayerClient 继承于 BnInterface<IMediaPlayerClient>，这是为基于 Android 的基础类 Binder 机制实现在进程间通信而构建的。其实根据 BnInterface 类模板的定义可知，类 BnInterface<IMediaPlayerClient>相当于双继承于 BnInterface 和 ImediaPlayerClient，这是 Android 一种常用的定义方式。

（2）头文件 mediaplayer.h

头文件 mediaplayer.h 是对外的接口类，主要功能是定义了一个 MediaPlayer 类，具体实现代码如下所示。

```
class MediaPlayer : public BnMediaPlayerClient,
                    public virtual IMediaDeathNotifier
{
public:
    MediaPlayer();
    ~MediaPlayer();
            void died();
            void disconnect();

            status_t setDataSource(
                    const char *url,
                    const KeyedVector<String8, String8> *headers);

            status_t setDataSource(int fd, int64_t offset, int64_t length);
            status_t setDataSource(const sp<IStreamSource> &source);
            status_t setVideoSurfaceTexture(
                                    const sp<IGraphicBufferProducer>& bufferProducer);
```

```
            status_t setListener(const sp<MediaPlayerListener>& listener);
            status_t prepare();
            status_t prepareAsync();
            status_t start();
            status_t stop();
            status_t pause();
            bool isPlaying();
            status_t getVideoWidth(int *w);
            status_t getVideoHeight(int *h);
            status_t seekTo(int msec);
            status_t getCurrentPosition(int *msec);
            status_t getDuration(int *msec);
            status_t reset();
            status_t setAudioStreamType(audio_stream_type_t type);
            status_t setLooping(int loop);
            bool isLooping();
            status_t setVolume(float leftVolume, float rightVolume);
            void notify(int msg, int ext1, int ext2, const Parcel *obj = NULL);
    static  status_t decode(const char* url, uint32_t *pSampleRate, int* pNumChannels,
                                    audio_format_t* pFormat,
                                    const sp<IMemoryHeap>& heap, size_t *pSize);
    static  status_t decode(int fd, int64_t offset, int64_t length, uint32_t *pSampleRate,
                                    int* pNumChannels, audio_format_t* pFormat,
                                    const sp<IMemoryHeap>& heap, size_t *pSize);
            status_t invoke(const Parcel& request, Parcel *reply);
            status_t setMetadataFilter(const Parcel& filter);
            status_t getMetadata(bool update_only, bool apply_filter, Parcel *metadata);
            status_t setAudioSessionId(int sessionId);
            int getAudioSessionId();
            status_t setAuxEffectSendLevel(float level);
            status_t attachAuxEffect(int effectId);
            status_t setParameter(int key, const Parcel& request);
            status_t getParameter(int key, Parcel* reply);
            status_t setRetransmitEndpoint(const char* addrString, uint16_t port);
            status_t setNextMediaPlayer(const sp<MediaPlayer>& player);

            status_t updateProxyConfig(
                    const char *host, int32_t port, const char *exclusionList);

private:
            void clear_l();
            status_t seekTo_l(int msec);
            status_t prepareAsync_l();
            status_t getDuration_l(int *msec);
            status_t attachNewPlayer(const sp<IMediaPlayer>& player);
            status_t reset_l();
            status_t doSetRetransmitEndpoint(const sp<IMediaPlayer>& player);

    sp<IMediaPlayer> mPlayer;
    thread_id_t mLockThreadId;
    Mutex mLock;
```

```
    Mutex mNotifyLock;
    Condition mSignal;
    sp<MediaPlayerListener> mListener;
    void* mCookie;
    media_player_states mCurrentState;
    int mCurrentPosition;
    int mSeekPosition;
    bool mPrepareSync;
    status_t mPrepareStatus;
    audio_stream_type_t mStreamType;
    bool mLoop;
    float mLeftVolume;
    float mRightVolume;
    int mVideoWidth;
    int mVideoHeight;
    int mAudioSessionId;
    float mSendLevel;
    struct sockaddr_in mRetransmitEndpoint;
    bool mRetransmitEndpointValid;
};

};
```

从上述接口代码中可以看出，类 MediaPlayer 刚好实现了一个 MediaPlayer 的基本操作，例如，播放（start）、停止（stop）、暂停（pause）等。

（3）头文件 IMediaDeathNotifier.h

类 DeathNotifier 继承了 IBinder 类中的 DeathRecipient 类，具体代码如下所示。

```
class IMediaDeathNotifier: virtual public RefBase
{
public:
    IMediaDeathNotifier() { addObitRecipient(this); }
    virtual ~IMediaDeathNotifier() { removeObitRecipient(this); }

    virtual void died() = 0;
    static const sp<IMediaPlayerService>& getMediaPlayerService();

private:
    IMediaDeathNotifier &operator=(const IMediaDeathNotifier &);
    IMediaDeathNotifier(const IMediaDeathNotifier &);

    static void addObitRecipient(const wp<IMediaDeathNotifier>& recipient);
    static void removeObitRecipient(const wp<IMediaDeathNotifier>& recipient);

    class DeathNotifier: public IBinder::DeathRecipient
    {
    public:
                DeathNotifier() {}
        virtual ~DeathNotifier();
```

```
        virtual void binderDied(const wp<IBinder>& who);
    };

    friend class DeathNotifier;

    static  Mutex sServiceLock;
    static  sp<IMediaPlayerService> sMediaPlayerService;
    static  sp<DeathNotifier> sDeathNotifier;
    static  SortedVector< wp<IMediaDeathNotifier> > sObitRecipients;
};

};
```

其实类 MediaPlayer 间接地继承了 IBinder，而 DeathNotifier 类继承了 IBinder::DeathRecipient，这都是为了实现进程间通信而构建的。

（4）头文件 IMediaPlayer.h

头文件 IMediaPlayer.h 的主要功能是实现 MediaPlayer 功能的接口，主要定义代码如下所示。

```
class IMediaPlayer: public IInterface
{
public:
    DECLARE_META_INTERFACE(MediaPlayer);

    virtual void disconnect() = 0;

    virtual status_t setDataSource(const char *url,
                                   const KeyedVector<String8, String8>* headers) = 0;
    virtual status_t setDataSource(int fd, int64_t offset, int64_t length) = 0;
    virtual status_t setDataSource(const sp<IStreamSource>& source) = 0;
    virtual status_t setVideoSurfaceTexture(
                                   const sp<IGraphicBufferProducer>& bufferProducer) = 0;
    virtual status_t prepareAsync() = 0;
    virtual status_t start() = 0;
    virtual status_t stop() = 0;
    virtual status_t pause() = 0;
    virtual status_t isPlaying(bool* state) = 0;
    virtual status_t seekTo(int msec) = 0;
    virtual status_t getCurrentPosition(int* msec) = 0;
    virtual status_t getDuration(int* msec) = 0;
    virtual status_t reset() = 0;
    virtual status_t setAudioStreamType(audio_stream_type_t type) = 0;
    virtual status_t setLooping(int loop) = 0;
    virtual status_t setVolume(float leftVolume, float rightVolume) = 0;
    virtual status_t setAuxEffectSendLevel(float level) = 0;
    virtual status_t attachAuxEffect(int effectId) = 0;
    virtual status_t setParameter(int key, const Parcel& request) = 0;
    virtual status_t getParameter(int key, Parcel* reply) = 0;
    virtual status_t setRetransmitEndpoint(const struct sockaddr_in* endpoint) = 0;
    virtual status_t getRetransmitEndpoint(struct sockaddr_in* endpoint) = 0;
    virtual status_t setNextPlayer(const sp<IMediaPlayer>& next) = 0;
```

在类 IMediaPlayer 中主要定义了 MediaPlayer 的功能接口，这个类必须被继承后才能够使用。需要注意的是，这些接口和类 MediaPlayer 的接口有些类似，但是它们并没有直接的关系。其实在类 MediaPlayer 的各种实现中，一般都会通过调用类 IMediaPlayer 的实现类来完成。

（5）头文件 IMediaPlayerService.h

头文件 IMediaPlayerService.h 的功能是描述一个 MediaPlayer 的服务，具体实现代码如下所示。

```
class IMediaPlayerService: public IInterface
{
public:
    DECLARE_META_INTERFACE(MediaPlayerService);

    virtual sp<IMediaRecorder> createMediaRecorder() = 0;
    virtual sp<IMediaMetadataRetriever> createMetadataRetriever() = 0;
    virtual sp<IMediaPlayer> create(const sp<IMediaPlayerClient>& client, int audioSessionId = 0) = 0;

    virtual status_t decode(const char* url, uint32_t *pSampleRate, int* pNumChannels,
                                        audio_format_t* pFormat,
                                        const sp<IMemoryHeap>& heap, size_t *pSize) = 0;
    virtual status_t decode(int fd, int64_t offset, int64_t length, uint32_t *pSampleRate,
                                        int* pNumChannels, audio_format_t* pFormat,
                                        const sp<IMemoryHeap>& heap, size_t *pSize) = 0;
    virtual sp<IOMX> getOMX() = 0;
    virtual sp<ICrypto> makeCrypto() = 0;
    virtual sp<IDrm> makeDrm() = 0;
    virtual sp<IHDCP> makeHDCP(bool createEncryptionModule) = 0;
    enum BatteryDataBits {
        kBatteryDataTrackAudio = 0x1,
        kBatteryDataTrackVideo = 0x2,
        kBatteryDataCodecStarted = 0x4,
        kBatteryDataTrackDecoder = 0x8,
        kBatteryDataAudioFlingerStart = 0x10,
        kBatteryDataAudioFlingerStop = 0x20,
        kBatteryDataSpeakerOn = 0x40,
        kBatteryDataOtherAudioDeviceOn = 0x80,
    };

    virtual void addBatteryData(uint32_t params) = 0;
    virtual status_t pullBatteryData(Parcel* reply) = 0;

    virtual status_t updateProxyConfig(
            const char *host, int32_t port, const char *exclusionList) = 0;
};
```

因为具有纯虚函数，所以 IMediaPlayerService 和 BnMediaPlayerService 必须被继承实现后才能够使用。在类 IMediaPlayerService 中定义的 create 和 decode 等接口，事实上是必须被继承者实现的内容。在此需要注意，create 的返回值的类型是 sp<IMediaPlayer>，这个 IMediaPlayer 正是提供实现功能的接口。

15.2.3　分析 Java 部分

在 Android 5.0 中，文件 packages/apps/Music/src/com/android/music/MediaPlaybackService.java 中包含了对 MediaPlayer 的调用。

在文件 MediaPlaybackService.java 中，通过 import 指令包含了对包 android.media.MediaPlayer 的引用，并且在类 MediaPlaybackService 的内部定义了类 MultiPlayer。文件 MediaPlaybackService.java 的主要实现代码如下所示。

```
public class MediaPlaybackService extends Service {
    public static final int NOW = 1;
    public static final int NEXT = 2;
    public static final int LAST = 3;
    public static final int PLAYBACKSERVICE_STATUS = 1;

    public static final int SHUFFLE_NONE = 0;
    public static final int SHUFFLE_NORMAL = 1;
    public static final int SHUFFLE_AUTO = 2;

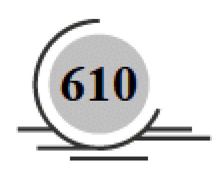
    public static final int REPEAT_NONE = 0;
    public static final int REPEAT_CURRENT = 1;
    public static final int REPEAT_ALL = 2;

    public static final String PLAYSTATE_CHANGED = "com.android.music.playstatechanged";
    public static final String META_CHANGED = "com.android.music.metachanged";
    public static final String QUEUE_CHANGED = "com.android.music.queuechanged";

    public static final String SERVICECMD = "com.android.music.musicservicecommand";
    public static final String CMDNAME = "command";
    public static final String CMDTOGGLEPAUSE = "togglepause";
    public static final String CMDSTOP = "stop";
    public static final String CMDPAUSE = "pause";
    public static final String CMDPLAY = "play";
    public static final String CMDPREVIOUS = "previous";
    public static final String CMDNEXT = "next";

    public static final String TOGGLEPAUSE_ACTION = "com.android.music.musicservicecommand.togglepause";
    public static final String PAUSE_ACTION = "com.android.music.musicservicecommand.pause";
    public static final String PREVIOUS_ACTION = "com.android.music.musicservicecommand.previous";
    public static final String NEXT_ACTION = "com.android.music.musicservicecommand.next";

    private static final int TRACK_ENDED = 1;
    private static final int RELEASE_WAKELOCK = 2;
    private static final int SERVER_DIED = 3;
    private static final int FOCUSCHANGE = 4;
    private static final int FADEDOWN = 5;
    private static final int FADEUP = 6;
    private static final int TRACK_WENT_TO_NEXT = 7;
    private static final int MAX_HISTORY_SIZE = 100;
```



```
private MultiPlayer mPlayer;
private String mFileToPlay;
private int mShuffleMode = SHUFFLE_NONE;
private int mRepeatMode = REPEAT_NONE;
private int mMediaMountedCount = 0;
private long [] mAutoShuffleList = null;
private long [] mPlayList = null;
private int mPlayListLen = 0;
private Vector<Integer> mHistory = new Vector<Integer>(MAX_HISTORY_SIZE);
private Cursor mCursor;
private int mPlayPos = -1;
private int mNextPlayPos = -1;
private static final String LOGTAG = "MediaPlaybackService";
private final Shuffler mRand = new Shuffler();
private int mOpenFailedCounter = 0;
String[] mCursorCols = new String[] {
        "audio._id AS _id",
        MediaStore.Audio.Media.ARTIST,
        MediaStore.Audio.Media.ALBUM,
        MediaStore.Audio.Media.TITLE,
        MediaStore.Audio.Media.DATA,
        MediaStore.Audio.Media.MIME_TYPE,
        MediaStore.Audio.Media.ALBUM_ID,
        MediaStore.Audio.Media.ARTIST_ID,
        MediaStore.Audio.Media.IS_PODCAST,
        MediaStore.Audio.Media.BOOKMARK
};
private final static int IDCOLIDX = 0;
private final static int PODCASTCOLIDX = 8;
private final static int BOOKMARKCOLIDX = 9;
private BroadcastReceiver mUnmountReceiver = null;
private WakeLock mWakeLock;
private int mServiceStartId = -1;
private boolean mServiceInUse = false;
private boolean mIsSupposedToBePlaying = false;
private boolean mQuietMode = false;
private AudioManager mAudioManager;
private boolean mQueueIsSaveable = true;
private boolean mPausedByTransientLossOfFocus = false;

private SharedPreferences mPreferences;
private int mCardId;

private MediaAppWidgetProvider mAppWidgetProvider = MediaAppWidgetProvider.getInstance();

private static final int IDLE_DELAY = 60000;

private RemoteControlClient mRemoteControlClient;
```

```
private Handler mMediaplayerHandler = new Handler() {
    float mCurrentVolume = 1.0f;
    @Override
    public void handleMessage(Message msg) {
        MusicUtils.debugLog("mMediaplayerHandler.handleMessage " + msg.what);
        switch (msg.what) {
            case FADEDOWN:
                mCurrentVolume -= .05f;
                if (mCurrentVolume > .2f) {
                    mMediaplayerHandler.sendEmptyMessageDelayed(FADEDOWN, 10);
                } else {
                    mCurrentVolume = .2f;
                }
                mPlayer.setVolume(mCurrentVolume);
                break;
            case FADEUP:
                mCurrentVolume += .01f;
                if (mCurrentVolume < 1.0f) {
                    mMediaplayerHandler.sendEmptyMessageDelayed(FADEUP, 10);
                } else {
                    mCurrentVolume = 1.0f;
                }
                mPlayer.setVolume(mCurrentVolume);
                break;
            case SERVER_DIED:
                if (mIsSupposedToBePlaying) {
                    gotoNext(true);
                } else {
                    openCurrentAndNext();
                }
                break;
            case TRACK_WENT_TO_NEXT:
                mPlayPos = mNextPlayPos;
                if (mCursor != null) {
                    mCursor.close();
                    mCursor = null;
                }
                if (mPlayPos >= 0 && mPlayPos < mPlayList.length) {
                    mCursor = getCursorForId(mPlayList[mPlayPos]);
                }
                notifyChange(META_CHANGED);
                updateNotification();
                setNextTrack();
                break;
            case TRACK_ENDED:
                if (mRepeatMode == REPEAT_CURRENT) {
                    seek(0);
                    play();
                } else {
                    gotoNext(false);
                }
```

```
                    break;
                case RELEASE_WAKELOCK:
                    mWakeLock.release();
                    break;

                case FOCUSCHANGE:
                    switch (msg.arg1) {
                        case AudioManager.AUDIOFOCUS_LOSS:
                            Log.v(LOGTAG, "AudioFocus: received AUDIOFOCUS_LOSS");
                            if(isPlaying()) {
                                mPausedByTransientLossOfFocus = false;
                            }
                            pause();
                            break;
                        case AudioManager.AUDIOFOCUS_LOSS_TRANSIENT_CAN_DUCK:
                            mMediaplayerHandler.removeMessages(FADEUP);
                            mMediaplayerHandler.sendEmptyMessage(FADEDOWN);
                            break;
                        case AudioManager.AUDIOFOCUS_LOSS_TRANSIENT:
                            Log.v(LOGTAG, "AudioFocus: received AUDIOFOCUS_LOSS_TRANSIENT");
                            if(isPlaying()) {
                                mPausedByTransientLossOfFocus = true;
                            }
                            pause();
                            break;
                        case AudioManager.AUDIOFOCUS_GAIN:
                            Log.v(LOGTAG, "AudioFocus: received AUDIOFOCUS_GAIN");
                            if(!isPlaying() && mPausedByTransientLossOfFocus) {
                                mPausedByTransientLossOfFocus = false;
                                mCurrentVolume = 0f;
                                mPlayer.setVolume(mCurrentVolume);
                                play();
                            } else {
                                mMediaplayerHandler.removeMessages(FADEDOWN);
                                mMediaplayerHandler.sendEmptyMessage(FADEUP);
                            }
                            break;
                        default:
                            Log.e(LOGTAG, "Unknown audio focus change code");
                    }
                    break;

                default:
                    break;
            }
        }
    };

    private BroadcastReceiver mIntentReceiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
```

```
            String action = intent.getAction();
            String cmd = intent.getStringExtra("command");
            MusicUtils.debugLog("mIntentReceiver.onReceive " + action + " / " + cmd);
            if (CMDNEXT.equals(cmd) || NEXT_ACTION.equals(action)) {
                gotoNext(true);
            } else if (CMDPREVIOUS.equals(cmd) || PREVIOUS_ACTION.equals(action)) {
                prev();
            } else if (CMDTOGGLEPAUSE.equals(cmd) || TOGGLEPAUSE_ACTION.equals(action)) {
                if (isPlaying()) {
                    pause();
                    mPausedByTransientLossOfFocus = false;
                } else {
                    play();
                }
            } else if (CMDPAUSE.equals(cmd) || PAUSE_ACTION.equals(action)) {
                pause();
                mPausedByTransientLossOfFocus = false;
            } else if (CMDPLAY.equals(cmd)) {
                play();
            } else if (CMDSTOP.equals(cmd)) {
                pause();
                mPausedByTransientLossOfFocus = false;
                seek(0);
            } else if (MediaAppWidgetProvider.CMDAPPWIDGETUPDATE.equals(cmd)) {
                int[] appWidgetIds = intent.getIntArrayExtra(AppWidgetManager.EXTRA_APPWIDGET_IDS);
                mAppWidgetProvider.performUpdate(MediaPlaybackService.this, appWidgetIds);
            }
        }
    };
…
private class MultiPlayer {
    private MediaPlayer mMediaPlayer = new MediaPlayer();
}
…
```

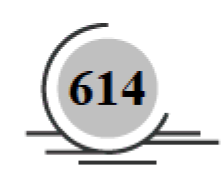
在类 MultiPlayer 中使用了类 MediaPlayer，实现了对这个 MediaPlayer 的调用，具体调用的过程如下所示。

```
mMediaPlayer.reset();
mMediaPlayer.setDataSource(path);
mMediaPlayer.setAudioStreamType(AudioManager.STREAM_MUSIC);
```

在上述代码中，接口 reset、setDataSource 和 setAudioStreamType 是通过 Java 本地调用（JNI）来实现的。

15.2.4　分析 JNI 部分

在 Android 5.0 中，MediaPlayer 的 Java 本地调用（JNI）部分通过文件 frameworks/base/media/jni/android_media_MediaPlayer.cpp 实现。



在文件 android_media_MediaPlayer.cpp 中定义了一个 JNINativeMethod（Java 本地调用方法）类型的数组 gMethods，具体代码如下所示。

```
static JNINativeMethod gMethods[] = {
    {
        "_setDataSource",
        "(Ljava/lang/String;[Ljava/lang/String;[Ljava/lang/String;)V",
        (void *)android_media_MediaPlayer_setDataSourceAndHeaders
    },

    {"_setDataSource", "(Ljava/io/FileDescriptor;JJ)V", (void *)android_media_MediaPlayer_setDataSourceFD},
    {"_setVideoSurface", "(Landroid/view/Surface;)V", (void *)android_media_MediaPlayer_setVideoSurface},
    {"prepare", "()V", (void *)android_media_MediaPlayer_prepare},
    {"prepareAsync", "()V", (void *)android_media_MediaPlayer_prepareAsync},
    {"_start", "()V", (void *)android_media_MediaPlayer_start},
    {"_stop", "()V", (void *)android_media_MediaPlayer_stop},
    {"getVideoWidth", "()I", (void *)android_media_MediaPlayer_getVideoWidth},
    {"getVideoHeight", "()I", (void *)android_media_MediaPlayer_getVideoHeight},
    {"seekTo", "(I)V", (void *)android_media_MediaPlayer_seekTo},
    {"_pause", "()V", (void *)android_media_MediaPlayer_pause},
    {"isPlaying", "()Z", (void *)android_media_MediaPlayer_isPlaying},
    {"getCurrentPosition", "()I", (void *)android_media_MediaPlayer_getCurrentPosition},
    {"getDuration", "()I", (void *)android_media_MediaPlayer_getDuration},
    {"_release", "()V", (void *)android_media_MediaPlayer_release},
    {"_reset", "()V", (void *)android_media_MediaPlayer_reset},
    {"setAudioStreamType", "(I)V", (void *)android_media_MediaPlayer_setAudioStreamType},
    {"setLooping", "(Z)V", (void *)android_media_MediaPlayer_setLooping},
    {"isLooping", "()Z", (void *)android_media_MediaPlayer_isLooping},
    {"setVolume", "(FF)V", (void *)android_media_MediaPlayer_setVolume},
    {"native_invoke", "(Landroid/os/Parcel;Landroid/os/Parcel;)I",(void *)android_media_MediaPlayer_invoke},
    {"native_setMetadataFilter", "(Landroid/os/Parcel;)I", (void *)android_media_MediaPlayer_setMetadataFilter},
    {"native_getMetadata", "(ZZLandroid/os/Parcel;)Z", (void *)android_media_MediaPlayer_getMetadata},
    {"native_init", "()V", (void *)android_media_MediaPlayer_native_init},
    {"native_setup", "(Ljava/lang/Object;)V", (void *)android_media_MediaPlayer_native_setup},
    {"native_finalize", "()V", (void *)android_media_MediaPlayer_native_finalize},
    {"getAudioSessionId", "()I", (void *)android_media_MediaPlayer_get_audio_session_id},
    {"setAudioSessionId", "(I)V", (void *)android_media_MediaPlayer_set_audio_session_id},
    {"setAuxEffectSendLevel", "(F)V", (void *)android_media_MediaPlayer_setAuxEffectSendLevel},
    {"attachAuxEffect", "(I)V", (void *)android_media_MediaPlayer_attachAuxEffect},
    {"native_pullBatteryData", "(Landroid/os/Parcel;)I", (void *)android_media_MediaPlayer_pullBatteryData},
    {"native_setRetransmitEndpoint", "(Ljava/lang/String;I)I", (void *)android_media_MediaPlayer_
setRetransmitEndpoint},
    {"setNextMediaPlayer", "(Landroid/media/MediaPlayer;)V", (void *)android_media_MediaPlayer_
setNextMediaPlayer},
    {"updateProxyConfig", "(Landroid/net/ProxyProperties;)V", (void *)android_media_MediaPlayer_
updateProxyConfig},
};
```

JNINativeMethod 成员的具体说明如下所示。

☑ 第一个成员是一个字符串，表示 Java 本地调用方法的名称，这个名称是在 Java 程序中调用的

名称。

☑ 第二个成员也是一个字符串，表示 Java 本地调用方法的参数和返回值。
☑ 第三个成员是 Java 本地调用方法对应的 C 语言函数。

其中，函数 android_media_MediaPlayer_reset()的具体实现代码如下所示。

```
static void
android_media_MediaPlayer_reset(JNIEnv *env, jobject thiz)
{
    ALOGV("reset");
    sp<MediaPlayer> mp = getMediaPlayer(env, thiz);
    if (mp == NULL ) {
        jniThrowException(env, "java/lang/IllegalStateException", NULL);
        return;
    }
    process_media_player_call( env, thiz, mp->reset(), NULL, NULL );
}
```

在对 android_media_MediaPlayer_reset()的调用过程中会得到一个 MediaPlayer 指针，通过对其调用实现具体的功能。register_android_media_MediaPlayer()能够将 gMethods 注册为类 android/media/MediaPlayer，具体实现代码如下所示。

```
static int register_android_media_MediaPlayer(JNIEnv *env)
{
    return AndroidRuntime::registerNativeMethods(env,
                "android/media/MediaPlayer", gMethods, NELEM(gMethods));
}
```

android/media/MediaPlayer 对应的 Java 类是 android.media.MediaPlayer。

下面是实现 Java 层接口 setDataSource 和 setAudioStreamType 的具体代码。

```
static void
android_media_MediaPlayer_setDataSourceAndHeaders(
        JNIEnv *env, jobject thiz, jstring path,
        jobjectArray keys, jobjectArray values) {

    sp<MediaPlayer> mp = getMediaPlayer(env, thiz);
    if (mp == NULL ) {
        jniThrowException(env, "java/lang/IllegalStateException", NULL);
        return;
    }

    if (path == NULL) {
        jniThrowException(env, "java/lang/IllegalArgumentException", NULL);
        return;
    }

    const char *tmp = env->GetStringUTFChars(path, NULL);
    if (tmp == NULL) {
        return;
    }
```

```
    ALOGV("setDataSource: path %s", tmp);

    String8 pathStr(tmp);
    env->ReleaseStringUTFChars(path, tmp);
    tmp = NULL;

    KeyedVector<String8, String8> headersVector;
    if (!ConvertKeyValueArraysToKeyedVector(
            env, keys, values, &headersVector)) {
        return;
    }

    status_t opStatus =
        mp->setDataSource(
                pathStr,
                headersVector.size() > 0? &headersVector : NULL);

    process_media_player_call(
            env, thiz, opStatus, "java/io/IOException",
            "setDataSource failed." );
}

static void
android_media_MediaPlayer_setDataSourceFD(JNIEnv *env, jobject thiz, jobject fileDescriptor, jlong offset,
jlong length)
{
    sp<MediaPlayer> mp = getMediaPlayer(env, thiz);
    if (mp == NULL ) {
        jniThrowException(env, "java/lang/IllegalStateException", NULL);
        return;
    }

    if (fileDescriptor == NULL) {
        jniThrowException(env, "java/lang/IllegalArgumentException", NULL);
        return;
    }
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    ALOGV("setDataSourceFD: fd %d", fd);
    process_media_player_call( env, thiz, mp->setDataSource(fd, offset, length), "java/io/IOException",
"setDataSourceFD failed." );
}
static void
android_media_MediaPlayer_setAudioStreamType(JNIEnv *env, jobject thiz, int streamtype)
{
    ALOGV("setAudioStreamType: %d", streamtype);
    sp<MediaPlayer> mp = getMediaPlayer(env, thiz);
    if (mp == NULL ) {
        jniThrowException(env, "java/lang/IllegalStateException", NULL);
        return;
    }
    process_media_player_call( env, thiz, mp->setAudioStreamType((audio_stream_type_t) streamtype) ,
```

```
NULL, NULL );
}
```

15.2.5 核心库 libmedia.so

在 Android 5.0 系统中，mediaplayer 的核心库是 libmedia.so，通过文件 frameworks/av/media/libmediamediaplayer.cpp 实现头文件 mediaplayer.h 提供的 4 个接口，主要实现代码如下所示。

```
status_t MediaPlayer::setDataSource(
        const char *url, const KeyedVector<String8, String8> *headers)
{
    ALOGV("setDataSource(%s)", url);
    status_t err = BAD_VALUE;
    if (url != NULL) {
        const sp<IMediaPlayerService>& service(getMediaPlayerService());
        if (service != 0) {
            sp<IMediaPlayer> player(service->create(this, mAudioSessionId));
            if ((NO_ERROR != doSetRetransmitEndpoint(player)) ||
                (NO_ERROR != player->setDataSource(url, headers))) {
                player.clear();
            }
            err = attachNewPlayer(player);
        }
    }
    return err;
}
status_t MediaPlayer::setDataSource(int fd, int64_t offset, int64_t length)
{
    ALOGV("setDataSource(%d, %lld, %lld)", fd, offset, length);
    status_t err = UNKNOWN_ERROR;
    const sp<IMediaPlayerService>& service(getMediaPlayerService());
    if (service != 0) {
        sp<IMediaPlayer> player(service->create(this, mAudioSessionId));
        if ((NO_ERROR != doSetRetransmitEndpoint(player)) ||
            (NO_ERROR != player->setDataSource(fd, offset, length))) {
            player.clear();
        }
        err = attachNewPlayer(player);
    }
    return err;
}

status_t MediaPlayer::setDataSource(const sp<IStreamSource> &source)
{
    ALOGV("setDataSource");
    status_t err = UNKNOWN_ERROR;
    const sp<IMediaPlayerService>& service(getMediaPlayerService());
    if (service != 0) {
        sp<IMediaPlayer> player(service->create(this, mAudioSessionId));
        if ((NO_ERROR != doSetRetransmitEndpoint(player)) ||
```

```
                (NO_ERROR != player->setDataSource(source))) {
                player.clear();
            }
            err = attachNewPlayer(player);
        }
        return err;
}
status_t MediaPlayer::setAudioStreamType(audio_stream_type_t type)
{
        ALOGV("MediaPlayer::setAudioStreamType");
        Mutex::Autolock _l(mLock);
        if (mStreamType == type) return NO_ERROR;
        if (mCurrentState & ( MEDIA_PLAYER_PREPARED | MEDIA_PLAYER_STARTED |
                    MEDIA_PLAYER_PAUSED | MEDIA_PLAYER_PLAYBACK_COMPLETE ) ) {
            ALOGE("setAudioStream called in state %d", mCurrentState);
            return INVALID_OPERATION;
        }
        mStreamType = type;
        return OK;
}
status_t MediaPlayer::reset()
{
        ALOGV("reset");
        Mutex::Autolock _l(mLock);
        return reset_l();
}
```

在函数 setDataSource()中，调用 getMediaPlayerService()得到了一个 IMediaPlayerService()，又从 IMediaPlayerService()中得到了 IMediaPlayer 类型的指针，通过这个指针实现具体的操作。其他一些函数的实现也与 setDataSource()类似。

另外，在 libmedia.so 中的其他文件与头文件的名称相同，分别是：

☑ IMediaPlayerClient.cpp

☑ IMediaPlayer.cpp

☑ IMediaPlayerService.cpp

为了实现 Binder 的具体功能，在上述类中还需要实现一个 BpXXX 的类，例如，文件 IMediaPlayerClient.cpp 的具体实现代码如下所示。

```
namespace android {

enum {
        NOTIFY = IBinder::FIRST_CALL_TRANSACTION,
};

class BpMediaPlayerClient: public BpInterface<IMediaPlayerClient>
{
public:
        BpMediaPlayerClient(const sp<IBinder>& impl)
            : BpInterface<IMediaPlayerClient>(impl)
        {
```

```
    }

    virtual void notify(int msg, int ext1, int ext2, const Parcel *obj)
    {
        Parcel data, reply;
        data.writeInterfaceToken(IMediaPlayerClient::getInterfaceDescriptor());
        data.writeInt32(msg);
        data.writeInt32(ext1);
        data.writeInt32(ext2);
        if (obj && obj->dataSize() > 0) {
            data.appendFrom(const_cast<Parcel *>(obj), 0, obj->dataSize());
        }
        remote()->transact(NOTIFY, data, &reply, IBinder::FLAG_ONEWAY);
    }
};

IMPLEMENT_META_INTERFACE(MediaPlayerClient, "android.media.IMediaPlayerClient");

// ----------------------------------------------------------------------

status_t BnMediaPlayerClient::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    switch (code) {
        case NOTIFY: {
            CHECK_INTERFACE(IMediaPlayerClient, data, reply);
            int msg = data.readInt32();
            int ext1 = data.readInt32();
            int ext2 = data.readInt32();
            Parcel obj;
            if (data.dataAvail() > 0) {
                obj.appendFrom(const_cast<Parcel *>(&data), data.dataPosition(), data.dataAvail());
            }

            notify(msg, ext1, ext2, &obj);
            return NO_ERROR;
        } break;
        default:
            return BBinder::onTransact(code, data, reply, flags);
    }
}

};
```

另外，还需要实现定义宏 IMPLEMENT_META_INTERFACE，展开这个宏后生成如下函数。

```
IMPLEMENT_META_INTERFACE(MediaPlayerClient, "android.hardware.IMediaPlayerClient");
```

上述过程都是基于 Binder 框架的实现方式，只需要按照模板实现即可完成。其中，形如 BpXXX 的类是代理类（proxy），形如 BnXXX 的类是本地类（native）。代理类的函数 transact()和本地类的函数 onTransact()用于实现对应的通信功能。



15.2.6　服务库 libmediaservice.so

在 Android 5.0 中，文件 frameworks/av/media/libmediaplayerservice/MediaPlayerService.h 和 frameworks/av/media/libmediaplayerservice/MediaPlayerService.cpp 用于实现一个 servers/media/ 服务，MediaPlayerService 是继承于 BnMediaPlayerService 的实现，在此类的内部又定义了类 Client，MediaPlayerService::Client 继承于 BnMediaPlayer。对应代码如下所示。

```
class MediaPlayerService : public BnMediaPlayerService
{
    class Client : public BnMediaPlayer
}
```

在 MediaPlayerService 中有一个如下所示的静态函数 instantiate()。

```
void MediaPlayerService::instantiate() {
    defaultServiceManager()->addService(
    String16("media.player"), new MediaPlayerService());
}
```

在函数 instantiate()中，调用 IServiceManager 函数 addService()向其中增加了一个名为 media.player 的服务。这个名为 media.player 的服务和文件 mediaplayer.cpp 中调用 getService()得到的名称一样。因此，在此通过调用 addService()增加服务时，可以在文件 mediaplayer.cpp 中按照名称 media.player 来使用。这个过程是使用 Binder 实现进程间的通信功能的，其实类 MediaPlayerService 是在服务中运行的，而文件 mediaplayer.cpp 调用功能在应用中运行，两者并不是一个进程。但是在文件 mediaplayer.cpp 中却像一个进程调用一样来调用 MediaPlayerService 的功能。

在文件 MediaPlayerService.cpp 中，函数 createPlayer()的具体实现代码如下所示。

```
static sp<MediaPlayerBase> createPlayer(player_type playerType, void* cookie, notify_callback_f notifyFunc)
{
    sp<MediaPlayerBase> p;
    switch (playerType) {
        case PV_PLAYER:
            LOGV(" create PVPlayer");
            p = new PVPlayer();
            break;
        case SONIVOX_PLAYER:
            LOGV(" create MidiFile");
            p = new MidiFile();
            break;
        case VORBIS_PLAYER:
            LOGV(" create VorbisPlayer");
            p = new VorbisPlayer();
            break;
    }
    //...
    return p;
}
```

在上述代码中，根据 playerType 的类型建立不同的播放器。在大多数情况下的类型是 PV_PLAYER，这时会调用 new PVPlayer()建立一个 PVPlayer，然后将其指针转换成 MediaPlayerBase 来使用。对于 Mini 文件来说，类型为 SONIVOX_PLAYER，此时将会建立一个 MidiFile。对于 Ogg Vorbis 格式的文件来说，将会建立一个 VorbisPlayer。

注意： OGG Vobis 是一种音频压缩格式，与 MP3 等音乐格式类似，具有完全免费、开放和没有专利限制的特点。

在 Android 系统中，类 PVPlayer、MidiFile 和 VorbisPlayer 都是通过继承 MediaPlayerInterface 得到的，而类 MediaPlayerInterface 又是通过继承 MediaPlayerBase 得到的，所以这 3 个类具有相同的接口类型。只有建立时会调用各自的构造函数，在建立之后只通过接口 MediaPlayerBase 来控制。

在 frameworks/base/media/libmediaplayerservice 目录中，通过文件 MidiFile.h 和 MidiFile.cpp 实现 MidiFile，通过文件 VorbisPlayer.h 和 VorbisPlayer.cpp 实现一个 VorbisPlayer。

15.2.7　OpenCorePlayer 实现 libopencoreplayer.so

在 Android 系统中，在 external/opencore/目录中实现 OpenCore Player，这个实现是一个基于 OpenCore 的 Player 的实现，具体实现的文件为 playerdriver.cpp。文件 playerdriver.cpp 实现了 PlayerDriver 和 PVPlayer 两个类。其中，PVPlayer 通过调用 PlayerDriver 的函数来实现具体的功能。

15.2.8　对 MediaPlayer 的总结

1. MediaPlayer 的状态

如图 15-7 所示为一个 MediaPlayer 对象被支持的播放控制操作驱动的生命周期和状态。其中，椭圆代表 MediaPlayer 对象可能驻留的状态，弧线表示驱动 MediaPlayer 在各个状态之间迁移的播放控制操作。这里有两种类型的弧线。由一个箭头开始的弧代表同步的方法调用，而以双箭头开头的弧线代表异步方法调用。

通过图 15-7 可以知道一个 MediaPlayer 对象有如下几种状态。

（1）当一个 MediaPlayer 对象被刚刚用 new 操作符创建或是调用了 reset()方法后，就处于 Idle 状态。当调用了 release()方法后，处于 End 状态。这两种状态之间是 MediaPlayer 对象的生命周期。

在一个新构建的 MediaPlayer 对象和一个调用了 reset()方法的 MediaPlayer 对象之间有一个微小但是十分重要的差别。在处于 Idle 状态时，调用 getCurrentPosition()、getDuration()、getVideoHeight()、getVideoWidth()、setAudioStreamType(int)、setLooping(boolean)、setVolume(float,float)、pause()、start()、stop()、seekTo(int)、prepare()或者 prepareAsync()方法都是编程错误。当一个 MediaPlayer 对象刚被构建时，内部的播放引擎和对象的状态都没有改变，这时调用以上方法，框架将无法回调客户端程序注册的 OnErrorListener.onError()方法；但若这个 MediaPlayer 对象调用了 reset()方法之后，再调用以上方法，内部的播放引擎就会回调客户端程序注册的 OnErrorListener.onError()方法，并将错误的状态传入。

笔者在此建议，一旦一个 MediaPlayer 对象不再被使用，应立即调用 release()方法来释放在内部的播放引擎中与这个 MediaPlayer 对象关联的资源。资源可能包括硬件加速组件的单态组件，若没有调用 release()方法可能会导致之后的 MediaPlayer 对象实例无法使用这种单态硬件资源，从而退回到软件实现

或运行失败。一旦 MediaPlayer 对象进入了 End 状态，将不能再被使用，也没有办法再迁移到其他状态。

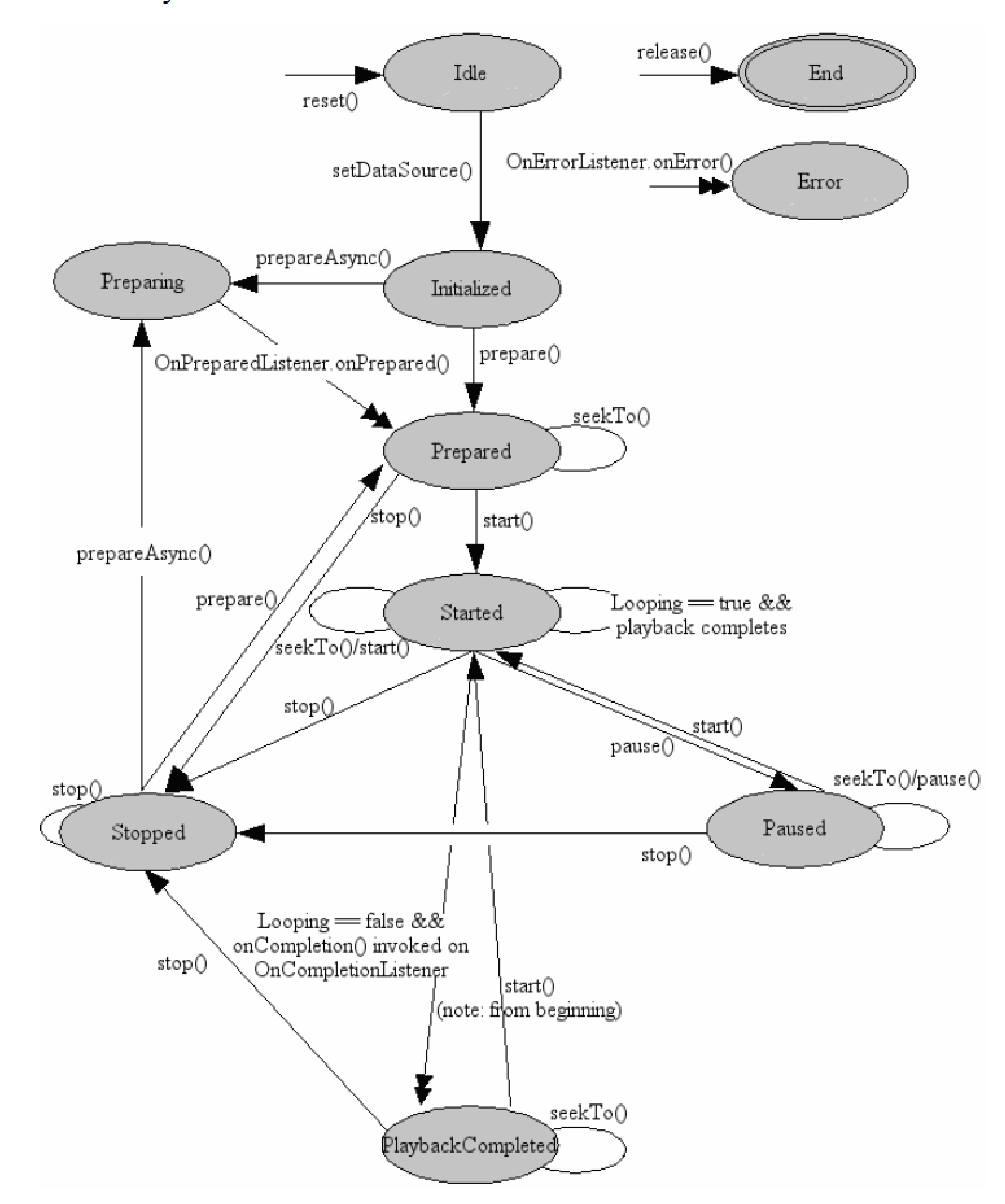


图 15-7　MediaPlayer 对象

此外，使用 new 操作符创建的 MediaPlayer 对象处于 Idle 状态，而那些通过重载的 create()便利方法创建的 MediaPlayer 对象却不是处于 Idle 状态。事实上，如果成功调用了重载的 create()方法，那么这些对象已经是 Prepare 状态了。

（2）在一般情况下，由于种种原因一些播放控制操作可能会失败，如不支持的音频/视频格式、缺少隔行扫描的音频/视频、分辨率太高、流超时等，因此，错误报告和恢复在这种情况下是非常重要的。有时由于编程错误，在处于无效状态的情况下调用了一个播放控制操作可能发生。在所有这些错误条件下，内部的播放引擎会调用一个由客户端程序员提供的 OnErrorListener.onError()方法。客户端程序员可以通过调用 MediaPlayer.setOnErrorListener(android.media.MediaPlayer.OnErrorListener)方法注册 OnErrorListener。

一旦发生错误，MediaPlayer 对象会进入 Error 状态。为了重用一个处于 Error 状态的 MediaPlayer 对象，可以调用 reset()方法把这个对象恢复成 Idle 状态。注册一个 OnErrorListener 来获知内部播放引擎发生的错误是好的编程习惯。在不合法的状态下调用一些方法，如 prepare()、prepareAsync()和 setDataSource()等会抛出 IllegalStateException 异常。

（3）调用 setDataSource(FileDescriptor)、setDataSource(String)、setDataSource(Context,Uri)或 setDataSource (FileDescriptor,long,long)方法会使处于 Idle 状态的对象迁移到 Initialized 状态。

若当此 MediaPlayer 处于其他状态下，调用 setDataSource()方法会抛出 IllegalStateException 异常。好的编程习惯是不要疏忽了调用 setDataSource()方法时可能抛出的 IllegalArgumentException 异常和 IOException 异常。

（4）在开始播放之前，MediaPlayer 对象必须要进入 Prepared 状态。

在此有如下两种方法（同步和异步）可以使 MediaPlayer 对象进入 Prepared 状态。

☑ 调用 prepare()方法（同步）：此方法返回就表示该 MediaPlayer 对象已经进入了 Prepared 状态。

☑ 调用 prepareAsync()方法（异步）：此方法会使此 MediaPlayer 对象进入 Preparing 状态并返回，内部的播放引擎会继续未完成的准备工作。

当同步版本返回或异步版本的准备工作完全完成时就会调用客户端程序员提供的 OnPreparedListener.onPrepared()监听方法。可以调用方法 MediaPlayer.setOnPreparedListener(android.media.MediaPlayer.OnPreparedListener)来注册 OnPreparedListener。

Preparing 是一个中间状态，如果在此状态下调用任何影响播放功能的方法，则最终的运行结果是未知的。在不合适的状态下调用 prepare()和 prepareAsync()方法会抛出 IllegalStateException 异常。当 MediaPlayer 对象处于 Prepared 状态时，可以调整音频/视频的属性，如音量、播放时是否一直亮屏、循环播放等。

（5）在要开始播放时必须调用 start()方法。当此方法成功返回时，MediaPlayer 的对象处于 Started 状态。isPlaying()方法可以被调用来测试某个 MediaPlayer 对象是否在 Started 状态。

当处于 Started 状态时，内部播放引擎会调用客户端程序员提供的 OnBufferingUpdateListener.onBufferingUpdate()回调方法，此回调方法允许应用程序追踪流播放的缓冲状态。对一个已经处于 Started 状态的 MediaPlayer 对象调用 start()方法没有影响。

（6）播放可以被暂停、停止以及调整当前播放位置。当调用 pause()方法并返回时，会使 MediaPlayer 对象进入 Paused 状态。注意 Started 与 Paused 状态的相互转换在内部的播放引擎中是异步的，所以可能需要一点时间在 isPlaying()方法中更新状态，若在播放流内容，这段时间可能会有几秒钟。

调用 start()方法会让一个处于 Paused 状态的 MediaPlayer 对象从之前暂停处恢复播放。当调用 start()方法返回时，MediaPlayer 对象的状态又会变成 Started 状态。对一个已经处于 Paused 状态的 MediaPlayer 对象，pause()方法没有影响。

（7）调用 stop()方法会停止播放，并且还会让一个处于 Started、Paused、Prepared 或 PlaybackCompleted 状态的 MediaPlayer 进入 Stopped 状态。对一个已经处于 Stopped 状态的 MediaPlayer 对象，stop()方法没有影响。

（8）调用 seekTo()方法可以调整播放的位置。方法 seekTo(int)是异步执行的，所以可以马上返回，但是实际的定位播放操作可能需要一段时间才能完成，尤其是播放流形式的音频/视频。当实际的定位播放操作完成之后，内部的播放引擎会调用客户端程序员提供的 OnSeekComplete.onSeekComplete()回调方法。可以通过 setOnSeekCompleteListener(OnSeekCompleteListener)方法注册。

在此需要注意，seekTo(int)方法也可以在其他状态下调用，例如 Prepared、Paused 和 PlaybackCompleted 状态。此外，目前的播放位置，实际可以调用 getCurrentPosition()方法得到，可以帮助如音乐播放器的应用程序不断更新播放进度。

（9）当播放到流的末尾时完成播放。如果调用 setLooping(boolean)方法开启了循环模式，那么这



个 MediaPlayer 对象会重新进入 Started 状态。如果没有开启循环模式，那么内部的播放引擎会调用客户端程序员提供的 OnCompletion.onCompletion()回调方法。可以通过调用 MediaPlayer.setOnCompletion Listener(OnCompletionListener)方法来设置。内部的播放引擎一旦调用了 OnCompletion.onCompletion()回调方法，说明这个 MediaPlayer 对象进入了 PlaybackCompleted 状态。当处于 PlaybackCompleted 状态时，可以再调用 start()方法来让这个 MediaPlayer 对象再进入 Started 状态。

2. MediaPlayer 方法的有效状态和无效状态

- ☑ getCurrentPosition {Idle, Initialized, Prepared, Started, Paused, Stopped, PlaybackCompleted} {Error}：在有效状态成功呼叫该方法不会改变此时的状态，在无效的状态呼叫该方法则会使该状态转换到错误状态中。
- ☑ getDuration {Prepared, Started, Paused, Stopped, PlaybackCompleted} {Idle, Initialized, Error}：在有效状态成功呼叫该方法不会改变此时的状态，在无效的状态呼叫该方法则会使该状态转换到错误状态中。
- ☑ getVideoHeight {Idle, Initialized, Prepared, Started, Paused, Stopped, PlaybackCompleted} {Error}：在有效状态成功呼叫该方法不会改变此时的状态，在无效的状态呼叫该方法则会使该状态转换到错误状态中。
- ☑ getVideoWidth {Idle, Initialized, Prepared, Started, Paused, Stopped, PlaybackCompleted} {Error}：在有效状态成功呼叫该方法不会改变此时的状态，在无效的状态呼叫该方法则会使该状态转换到错误状态中。
- ☑ isPlaying {Idle, Initialized, Prepared, Started, Paused, Stopped, PlaybackCompleted} {Error}：在有效状态成功呼叫该方法不会改变此时的状态，在无效的状态呼叫该方法则会使该状态转换到错误状态中。
- ☑ pause {Started, Paused} {Idle, Initialized, Prepared, Stopped, PlaybackCompleted, Error}：在有效状态成功呼叫该方法改变此时的状态到暂停状态，在无效的状态呼叫该方法则会使该状态转换到错误状态中。
- ☑ prepare {Initialized, Stopped} {Idle, Prepared, Started, Paused, PlaybackCompleted, Error}：在有效状态成功呼叫该方法改变此时的状态到准备状态，在无效的状态呼叫该方法则会抛出错误状态异常。
- ☑ prepareAsync {Initialized, Stopped} {Idle, Prepared, Started, Paused, PlaybackCompleted, Error}：在有效状态成功呼叫该方法改变此时的状态到准备状态，在无效的状态呼叫该方法则会抛出错误状态异常。
- ☑ release any {}：在调用 release()后该对象不再是可用的。
- ☑ reset {Idle, Initialized, Prepared, Started, Paused, Stopped, PlaybackCompleted, Error} {}：在调用 reset()后该对象如刚创建的一样。
- ☑ seekTo {Prepared, Started, Paused, PlaybackCompleted} {Idle, Initialized, Stopped, Error}：在有效状态成功呼叫该方法改变此时的状态到暂停状态，在无效的状态呼叫该方法则会使该状态转换到错误状态中。
- ☑ setAudioStreamType {Idle, Initialized, Stopped, Prepared, Started, Paused, PlaybackCompleted} {Error}：在有效状态成功呼叫该方法改变此时的状态到暂停状态。

☑ setDataSource {Idle} {Initialized, Prepared, Started, Paused, Stopped, PlaybackCompleted, Error}：在有效状态成功呼叫该方法改变此时的状态到初始化状态，在无效的状态呼叫该方法则会抛出错误状态异常。
☑ setDisplay any {}：在任何状态都可以呼叫该方法且不会改变当前对象的状态。
☑ setLooping {Idle, Initialized, Stopped, Prepared, Started, Paused, PlaybackCompleted} {Error}：在有效状态成功呼叫该方法不会改变此时的状态，在无效的状态呼叫该方法则会使该状态转换到错误状态中。
☑ isLooping any {}：在任何状态都可以呼叫该方法且不会改变当前对象的状态。
☑ setOnBufferingUpdateListener any {}：在任何状态都可以呼叫该方法且不会改变当前对象的状态。
☑ setOnCompletionListener any {}：在任何状态都可以呼叫该方法且不会改变当前对象的状态。
☑ setOnErrorListener any {}：在任何状态都可以呼叫该方法且不会改变当前对象的状态。
☑ setOnPreparedListener any {}：在任何状态都可以呼叫该方法且不会改变当前对象的状态。
☑ setOnSeekCompleteListener any {}：在任何状态都可以呼叫该方法且不会改变当前对象的状态。
☑ setScreenOnWhilePlaying any {}：在任何状态都可以呼叫该方法且不会改变当前对象的状态。
☑ setVolume {Idle, Initialized, Stopped, Prepared, Started, Paused, PlaybackCompleted} {Error}：成功调用该方法不会改变当前的状态。
☑ setWakeMode any {}：在任何状态都可以呼叫该方法且不会改变当前对象的状态。
☑ start {Prepared, Started, Paused, PlaybackCompleted} {Idle, Initialized, Stopped, Error}：在有效状态成功呼叫该方法改变此时的状态到开始状态，在无效的状态呼叫该方法则会转换到错误状态。
☑ stop {Prepared, Started, Stopped, Paused, PlaybackCompleted} {Idle, Initialized, Error}：在有效状态成功呼叫该方法改变此时的状态到停止状态，在无效的状态呼叫该方法则会转换到错误状态。

3．MediaPlayer 方法的接口

☑ 接口 MediaPlayer.OnBufferingUpdateListener：定义了唤起指明网络上的媒体资源以缓冲流的形式播放。
☑ 接口 MediaPlayer.OnCompletionListener：是为当媒体资源的播放完成后被唤起的回放定义的。
☑ 接口 MediaPlayer.OnErrorListener：定义了当在异步操作时（其他错误将会在呼叫方法时抛出异常）出现错误后唤起的回放操作。
☑ 接口 MediaPlayer.OnInfoListener：定义了与一些关于媒体或播放的信息以及警告相关的被唤起的回放。
☑ 接口 MediaPlayer.OnPreparedListener：定义为媒体的资源准备播放时唤起回放准备的。
☑ 接口 MediaPlayer.OnSeekCompleteListener：定义了指明查找操作完成后唤起的回放操作。
☑ 接口 MediaPlayer.OnVideoSizeChangedListener：定义了当视频大小被首次知晓或更新时唤起的回放。

4．MediaPlayer 方法的常量

☑ int MEDIA_ERROR_NOT_VALID_FOR_PROGRESSIVE_PLAYBACK：视频流及其容器时不支持连续的非处于播放文件内的播放视频序列。

- ☑ int MEDIA_ERROR_SERVER_DIED：媒体服务终止。
- ☑ int MEDIA_ERROR_UNKNOWN：未指明的媒体播放错误。
- ☑ int MEDIA_INFO_BAD_INTERLEAVING：不正确的交叉存储技术意味着媒体被不适当的交叉存储或者根本就没有交叉存储。
- ☑ int MEDIA_INFO_METADATA_UPDATE：一套新的可用的元数据。
- ☑ int MEDIA_INFO_NOT_SEEKABLE：媒体位置不可查找。
- ☑ int MEDIA_INFO_UNKNOWN：未指明的媒体播放信息。
- ☑ int MEDIA_INFO_VIDEO_TRACK_LAGGING：视频相对于解码器太复杂以至于不能解码足够快的帧率。

5. MediaPlayer 方法的公共方法

- ☑ static MediaPlayer create(Context context, Uri uri)：根据给定的 uri 方便地创建 MediaPlayer 对象的方法。
- ☑ static MediaPlayer create(Context context, int resid) ：根据给定的资源 id 方便地创建 MediaPlayer 对象的方法。
- ☑ static MediaPlayer create(Context context, Uri uri, SurfaceHolder holder) ：根据给定的 uri 方便地创建 MediaPlayer 对象的方法。
- ☑ int getCurrentPosition()：获得当前播放的位置。
- ☑ int getDuration()：获得文件段。
- ☑ int getVideoHeight()：获得视频的高度。
- ☑ int getVideoWidth()：获得视频的宽度。
- ☑ boolean isLooping()：检查 MedioPlayer 是否处于循环。
- ☑ boolean isPlaying()：检查 MedioPlayer 是否在播放。
- ☑ void pause()：暂停播放。
- ☑ void prepare()：让播放器处于准备状态（同步的）。
- ☑ void prepareAsync()：让播放器处于准备状态（异步的）。
- ☑ void release()：释放与 MediaPlayer 相关的资源。
- ☑ void reset()：重置 MediaPlayer 到初始化状态。
- ☑ void seekTo(int msec)：搜寻指定的时间位置。
- ☑ void setAudioStreamType(int streamtype)：为 MediaPlayer 设定音频流类型。
- ☑ void setDataSource(String path)：指定的 path 路径所代表的文件。
- ☑ void setDataSource(FileDescriptor fd, long offset, long length)：指定装载 fd 所代表的文件中从 offset 开始、长度为 length 的文件内容。
- ☑ void setDataSource(FileDescriptor fd)：设定使用的数据源（filedescriptor）。
- ☑ void setDataSource(Context context, Uri uri)：设定一个如 Uri 内容的数据源。
- ☑ void setDisplay(SurfaceHolder sh)：设定播放该 Video 的媒体播放器的 SurfaceHolder。
- ☑ void setLooping(boolean looping)：设定播放器循环或是不循环。
- ☑ void setOnBufferingUpdateListener(MediaPlayer.OnBufferingUpdateListener listener)：注册一个当网络缓冲数据流变化时唤起的播放事件。

- ☑ void setOnCompletionListener(MediaPlayer.OnCompletionListener listener)：注册一个当媒体资源在播放到达终点时唤起的播放事件。
- ☑ void setOnErrorListener(MediaPlayer.OnErrorListener listener)：注册一个当在异步操作过程中发生错误时唤起的播放事件。
- ☑ void setOnInfoListener(MediaPlayer.OnInfoListener listener)：注册一个当有信息/警告出现时唤起的播放事件。
- ☑ void setOnPreparedListener(MediaPlayer.OnPreparedListener listener)：注册一个当媒体资源准备播放时唤起的播放事件。
- ☑ void setOnSeekCompleteListener(MediaPlayer.OnSeekCompleteListener listener)：注册一个当搜寻操作完成后唤起的播放事件。
- ☑ void setOnVideoSizeChangedListener(MediaPlayer.OnVideoSizeChangedListener listener) ：注册一个当视频大小知晓或更新后唤起的播放事件。
- ☑ void setScreenOnWhilePlaying(boolean screenOn)：控制当视频播放发生时是否使用 SurfaceHolder 来保持屏幕。
- ☑ void setVolume(float leftVolume, float rightVolume)：设置播放器的音量。
- ☑ void setWakeMode(Context context, int mode)：为 MediaPlayer 设置低等级的电源管理状态。
- ☑ void start()：开始或恢复播放。
- ☑ void stop()：停止播放。

15.3 VideoView 详解

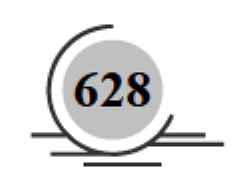
在 Android 系统中，内置了 VideoView Widget 作为多媒体视频播放器。VideoView 的用法和其他 Android 中 Widget 私用方法类似。在使用 VideoView 时，必须先在 Layout XML 中定义 VideoView 属性，然后在程序中通过 findViewById()方法即可创建 VideoView 对象。VideoView 的最大用处是播放视频文件，类 VideoView 可以从不同的来源（例如资源文件或内容提供器）读取图像，计算和维护视频的画面尺寸以使其适用于任何布局管理器，并提供一些诸如缩放、着色之类的显示选项。在 Android 5.0 中，VideoView 的实现文件是 frameworks/base/core/java/android/widget/VideoView.java。本节将详细讲解使用 VideoView 播放视频的基本知识，为读者学习本书后面的知识打下基础。

15.3.1 构造函数

在类 VideoView 中有 3 个构造函数，其中，第一个构造函数的实现代码如下所示。

```
public VideoView(Context context) {
	super(context);
	initVideoView();
}
```

通过上述函数可以创建一个默认属性的 VideoView 实例，参数 context 表示视图运行的应用程序上下文，通过此参数可以访问当前主题、资源等。



第二个构造函数的实现代码如下所示。

```
public VideoView(Context context, AttributeSet attrs) {
      this(context, attrs, 0);
      initVideoView();
}
```

通过上述函数可以创建一个带有 attrs 属性的 VideoView 实例，各个参数的具体说明如下所示。

☑ context：表示视图运行的应用程序上下文，通过此参数可以访问当前主题、资源等。

☑ attrs：用于视图的 XML 标签属性集合。

第三个构造函数的实现代码如下所示。

```
public VideoView(Context context, AttributeSet attrs, int defStyle) {
      super(context, attrs, defStyle);
      initVideoView();
}
```

通过上述函数可以创建一个带有 attrs 属性，并且指定其默认样式的 VideoView 实例。各个参数的具体说明如下所示。

☑ context：视图运行的应用程序上下文，通过此参数可以访问当前主题、资源等。

☑ attrs：用于视图的 XML 标签属性集合。

☑ defStyle：应用到视图的默认风格。如果为 0 则不应用（包括当前主题中的）风格。该值可以是当前主题中的属性资源，或者是明确的风格资源 ID。

15.3.2 公共方法

在类 VideoView 中，包含了如下公共方法。

（1）public boolean canPause()：判断是否能够暂停播放视频，具体实现代码如下所示。

```
@Override
public boolean canPause() {
      return mCanPause;
}
```

（2）public boolean canSeekBackward()：判断是否能够倒退，具体实现代码如下所示。

```
@Override
public boolean canSeekBackward() {
      return mCanSeekBack;
}
```

（3）public boolean canSeekForward()：判断是否能够快进，具体实现代码如下所示。

```
@Override
public boolean canSeekForward() {
      return mCanSeekForward;
}
```

（4）public int getBufferPercentage()：获得缓冲区的百分比，具体实现代码如下所示。

```
@Override
public int getBufferPercentage() {
    if (mMediaPlayer != null) {
        return mCurrentBufferPercentage;
    }
    return 0;
}
```

（5）public int getCurrentPosition()：获得当前的位置，具体实现代码如下所示。

```
@Override
public int getCurrentPosition() {
    if (isInPlaybackState()) {
        return mMediaPlayer.getCurrentPosition();
    }
    return 0;
}
```

（6）public int getDuration()：获得所播放视频的总时间，具体实现代码如下所示。

```
@Override
public int getDuration() {
    if (isInPlaybackState()) {
        return mMediaPlayer.getDuration();
    }

    return -1;
}
```

（7）public boolean isPlaying()：判断是否正在播放视频，具体实现代码如下所示。

```
@Override
public boolean isPlaying() {
    return isInPlaybackState() && mMediaPlayer.isPlaying();
}
```

（8）public boolean onKeyDown(int keyCode, KeyEvent event)：是 KeyEvent.Callback.onKeyMultiple()的默认实现。如果视图可用并可按，当触发 KEYCODE_DPAD_CENTER 或 KEYCODE_ENTER 时执行视图的按下事件。如果处理了事件则返回 true；如果允许下一个事件接受器处理该事件则返回 false。函数 onKeyDown()的具体实现代码如下所示。

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event)
{
    boolean isKeyCodeSupported = keyCode != KeyEvent.KEYCODE_BACK &&
                                 keyCode != KeyEvent.KEYCODE_VOLUME_UP &&
                                 keyCode != KeyEvent.KEYCODE_VOLUME_DOWN &&
                                 keyCode != KeyEvent.KEYCODE_VOLUME_MUTE &&
                                 keyCode != KeyEvent.KEYCODE_MENU &&
                                 keyCode != KeyEvent.KEYCODE_CALL &&
                                 keyCode != KeyEvent.KEYCODE_ENDCALL;
    if (isInPlaybackState() && isKeyCodeSupported && mMediaController != null) {
```

```
            if (keyCode == KeyEvent.KEYCODE_HEADSETHOOK ||
                    keyCode == KeyEvent.KEYCODE_MEDIA_PLAY_PAUSE) {
                if (mMediaPlayer.isPlaying()) {
                    pause();
                    mMediaController.show();
                } else {
                    start();
                    mMediaController.hide();
                }
                return true;
            } else if (keyCode == KeyEvent.KEYCODE_MEDIA_PLAY) {
                if (!mMediaPlayer.isPlaying()) {
                    start();
                    mMediaController.hide();
                }
                return true;
            } else if (keyCode == KeyEvent.KEYCODE_MEDIA_STOP
                    || keyCode == KeyEvent.KEYCODE_MEDIA_PAUSE) {
                if (mMediaPlayer.isPlaying()) {
                    pause();
                    mMediaController.show();
                }
                return true;
            } else {
                toggleMediaControlsVisiblity();
            }
        }
        return super.onKeyDown(keyCode, event);
}
```

各个参数的具体说明如下所示。

☑ keyCode：表示按下的键在 KEYCODE_ENTER 中定义的键盘代码。

☑ event：KeyEvent 对象，定义了按钮动作。

（9）public boolean onTouchEvent(MotionEvent ev)：通过该方法来处理触屏事件，参数 event 表示触屏事件。如果事件已经处理则返回 true，否则返回 false。函数 onTouchEvent()的具体实现代码如下所示。

```
@Override
public boolean onTouchEvent(MotionEvent ev) {
    if (isInPlaybackState() && mMediaController != null) {
        toggleMediaControlsVisiblity();
    }
    return false;
}
```

（10）public boolean onTrackballEvent(MotionEvent ev)：实现这个方法去处理轨迹球的动作事件，轨迹球相对于上次事件移动的位置能用 MotionEvent.getX() 和 MotionEvent.getY()函数取回。当用户按下方向键时，将被作为一次移动操作来处理（为了表现来自轨迹球的更小粒度的移动信息，返回小数）。函数 onTrackballEvent()的具体实现代码如下所示。

```
@Override
public boolean onTrackballEvent(MotionEvent ev) {
```

```
    if (isInPlaybackState() && mMediaController != null) {
        toggleMediaControlsVisiblity();
    }
    return false;
}
```

参数 ev 表示动作的事件。

（11）public void pause()：使得播放暂停，具体实现代码如下所示。

```
@Override
public void pause() {
    if (isInPlaybackState()) {
        if (mMediaPlayer.isPlaying()) {
            mMediaPlayer.pause();
            mCurrentState = STATE_PAUSED;
        }
    }
    mTargetState = STATE_PAUSED;
}
```

（12）public int resolveAdjustedSize(int desiredSize, int measureSpec)：取得调整后的尺寸。如果 measureSpec 对象传入的模式是 UNSPECIFIED，那么返回的是 desiredSize。函数 resolveAdjustedSize() 的具体实现代码如下所示。

```
public int resolveAdjustedSize(int desiredSize, int measureSpec) {
    return getDefaultSize(desiredSize, measureSpec);
}
```

如果 measureSpec 对象传入的模式是 AT_MOST，返回的将是 desiredSize 和 measureSpec 对象的尺寸中最小的那个。如果 measureSpec 对象传入的模式是 EXACTLY，那么返回的是 measureSpec 对象中的尺寸大小值。

注意： MeasureSpec 是一个 android.view.View 的内部类，封装了从父类传送到子类的布局要求信息。每个 MeasureSpec 对象描述了控件的高度或者宽度。MeasureSpec 对象是由尺寸和模式组成的，有 3 个模式：UNSPECIFIED、EXACTLY 和 AT_MOST，这个对象由 MeasureSpec.makeMeasureSpec() 函数创建。

（13）public void resume()：用于恢复挂起的播放器，具体实现代码如下所示。

```
public void resume() {
    openVideo();
}
```

（14）public void seekTo(int msec)：设置播放位置，具体实现代码如下所示。

```
@Override
public void seekTo(int msec) {
    if (isInPlaybackState()) {
        mMediaPlayer.seekTo(msec);
        mSeekWhenPrepared = 0;
```

```
    } else {
        mSeekWhenPrepared = msec;
    }
}
```

（15）public void setMediaController(MediaController controller)：设置媒体控制器，具体实现代码如下所示。

```
public void setMediaController(MediaController controller) {
    if (mMediaController != null) {
        mMediaController.hide();
    }
    mMediaController = controller;
    attachMediaController();
}
```

（16）public void setOnCompletionListener(MediaPlayer.OnCompletionListener l)：注册在媒体文件播放完毕时调用的回调函数。具体实现代码如下所示。

```
public void setOnCompletionListener(OnCompletionListener l)
{
    mOnCompletionListener = l;
}
```

参数 l 表示要执行的回调函数。

（17）public void setOnErrorListener(MediaPlayer.OnErrorListener l)：注册在设置或播放过程中发生错误时调用的回调函数。具体实现代码如下所示。

```
public void setOnErrorListener(OnErrorListener l)
{
    mOnErrorListener = l;
}
```

如果未指定回调函数，或回调函数返回 false，VideoView 会通知用户发生了错误。参数 l 表示要执行的回调函数。

（18）public void setOnPreparedListener(MediaPlayer.OnPreparedListener l)：用于注册在媒体文件加载完毕，可以播放时调用的回调函数。具体实现代码如下所示。

```
public void setOnPreparedListener(MediaPlayer.OnPreparedListener l)
{
    mOnPreparedListener = l;
}
```

参数 l 表示要执行的回调函数。

（19）public void setVideoPath(String path)：用于设置视频文件的路径名，具体实现代码如下所示。

```
public void setVideoPath(String path) {
    setVideoURI(Uri.parse(path));
}
```

（20）public void setVideoURI(Uri uri)：设置视频文件的统一资源标识符，具体实现代码如下所示。

```
public void setVideoURI(Uri uri, Map<String, String> headers) {
      mUri = uri;
      mHeaders = headers;
      mSeekWhenPrepared = 0;
      openVideo();
      requestLayout();
      invalidate();
}
```

（21）public void start()：开始播放视频文件，具体实现代码如下所示。

```
@Override
public void start() {
      if (isInPlaybackState()) {
            mMediaPlayer.start();
            mCurrentState = STATE_PLAYING;
      }
      mTargetState = STATE_PLAYING;
}
```

（22）public void stopPlayback()：停止回放视频文件，具体实现代码如下所示。

```
private Vector<Pair<InputStream, MediaFormat>> mPendingSubtitleTracks;
public void stopPlayback() {
      if (mMediaPlayer != null) {
            mMediaPlayer.stop();
            mMediaPlayer.release();
            mMediaPlayer = null;
            mCurrentState = STATE_IDLE;
            mTargetState    = STATE_IDLE;
      }
}
```

（23）public void suspend()：挂起视频文件的播放，具体实现代码如下所示。

```
public void suspend() {
      release(false);
}
```

第 16 章　WebKit 系统架构详解

随着互联网的兴起和发展，现在互联网已经成为了人们生活中必不可少的一部分。网上冲浪、QQ/微信交友聊天、天猫/京东购物等应用已经被广大用户所认识并接受，互联网在不知不觉间改变了人们的生活。Android 作为一款移动智能设备系统，自然具备访问互联网的功能。本章将详细讲解 Android 5.0 系统中 WebKit 系统的架构知识，为读者深入理解 Android 系统的架构打下基础。

16.1　WebKit 系统目录

在 Android 系统中，WebKit 是一个网络引擎库，其主要功能是实现 Web 浏览器的引擎，达到浏览网页数据的效果。在 Android 系统中，浏览 Web 浏览器和一个可嵌入的 Web 视图的功能是通过 Webkit 实现的，这是一个第三方开发的浏览器引擎。WebKit 的源码被保存在/external/webkit/目录下，其目录结构如下所示。

```
external/webkit/
├── Examples                        //WebKit 例子
├── LayoutTests                     //布局测试
├── PerformanceTests                //表现测试
├── Source                          //WebKit 源代码
├── Tools                           //工具
├── WebKitLibraries                 //WebKit 用到的库
├── Android.mk                      //Makefile
├── bison_check.mk
├── CleanSpec.mk
├── MODULE_LICENSE_LGPL             //证书
├── NOTICE
└── WEBKIT_MERGE_REVISION           //版本信息
```

为了从更深的层次了解 WebKit 浏览器编程的基本知识，本书将首先从 Android 底层开始分析 WebKit 系统的原理和用法，依次从下到上分析 WebKit 浏览器编程的基本知识。在 Android 系统中，WebKit 模块分成 Java 和 WebKit 库两个部分，具体说明如下所示。

☑ Java 层：负责与 Android 应用程序通信。

☑ WebKit 类库：因为是由 C/C++实现的，所以也被称为 C 层库，WebKit 类库部分负责实际的网页排版处理。

Java 层和 WebKit 类库之间通过 JNI 和 Bridge 实现相互调用，如图 16-1 所示。

本节将详细分析 Android 5.0 系统中 WebKit 网络引擎库的基本源码。

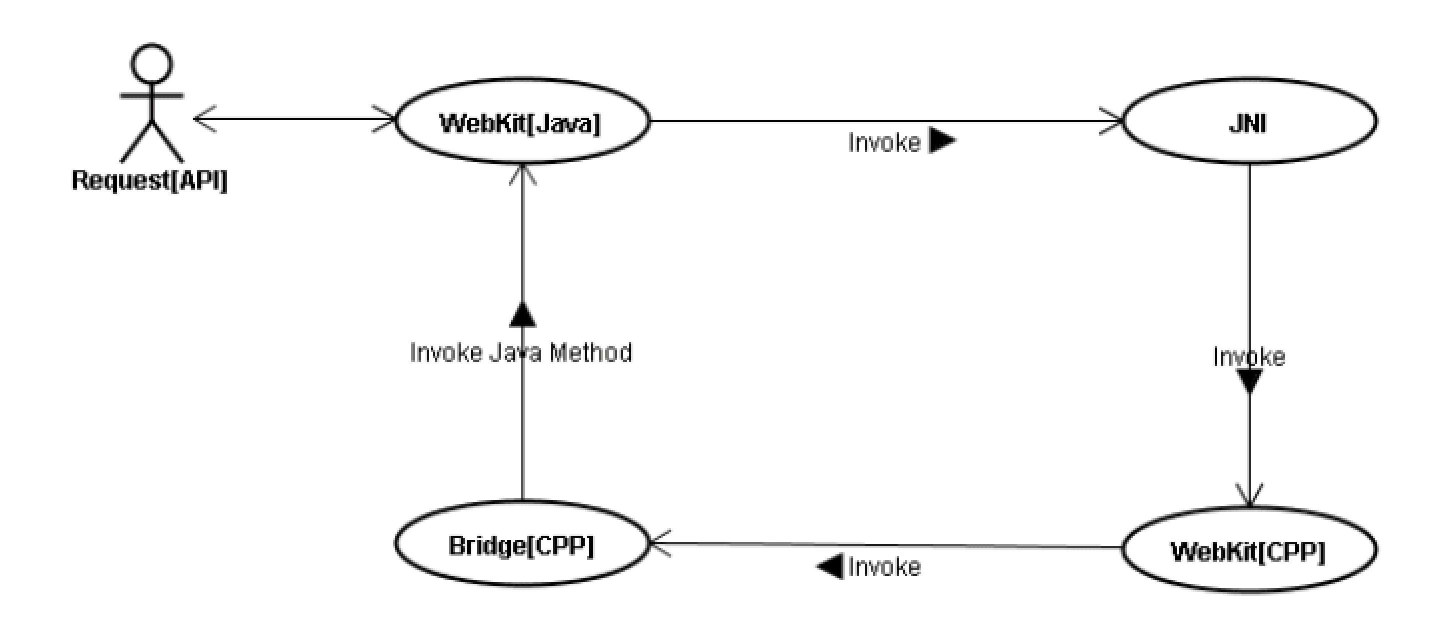


图 16-1　WebKit 系统框架结构

16.2　Java 层的基本框架

在 Android 系统中，WebKit 模块中 Java 层的根目录是/frameworks/base/core/java/android/webkit/。上述目录是基于 Android 5.0 的，其目录结构如表 16-1 所示。

表 16-1　WebKit 的目录结构

目　　录	说　　明
BrowserFrame.java	BrowserFrame 对象是对 WebCore 库中的 Frame 对象的 Java 层封装，用于创建 WebCore 中定义的 Frame，以及为该 Frame 对象提供 Java 层回调方法
ByteArrayBuilder.java	ByteArrayBuilder 辅助对象，用于 byte 块链表的处理
CachLoader.java	URL Cache 载入器对象，该对象实现 StreadLoader 抽象基类，用于通过 CacheResult 对象载入内容数据
CacheManager.java	Cache 管理对象，负责 Java 层 Cache 对象管理
CacheSyncManager.java	Cache 同步管理对象，负责同步 RAM 和 Flash 之间的浏览器 Cache 数据。实际的物理数据操作在 WebSyncManager 对象中完成
CallbackProxy.java	该对象是用于处理 WebCore 与 UI 线程消息的代理类。当有 Web 事件产生时 WebCore 线程会调用该回调代理类，代理类会通过消息的方式通知 UI 线程，并且调用设置的客户对象的回调函数
CellList.java	CellList 定义图片集合中的 Cell，管理 Cell 图片的绘制、状态改变以及索引
CookieManager.java	根据 RFC2109 规范来管理 Cookies
CookieSyncManager.java	Cookies 同步管理对象，该对象负责同步 RAM 和 Flash 之间的 Cookies 数据。实际的物理数据操作在基类 WebSyncManager 中完成
DataLoader.java	数据载入器对象，用于载入网页数据
DateSorter.java	尚未使用
DownloadListener.java	下载侦听器接口
DownloadManagerCore.java	下载管理器对象，管理下载列表。该对象运行在 WebKit 的线程中，通过 CallbackProxy 对象与 UI 线程交互
FileLoader.java	文件载入器，将文件数据载入到 Frame 中
FrameLoader.java	Frame 载入器，用于载入网页 Frame 数据
HttpAuthHandler.java	Http 认证处理对象，该对象会作为参数传递给 BrowserCallback.displayHttpAuthDialog 方法，与用户交互

续表

目　录	说　明
HttpDataTime.java	该对象是处理 HTTP 日期的辅助对象
JSConfirmResult.java	JS 确认请求对象
JSPromptResult.java	JS 结果提示对象，用于向用户提示 JavaScript 运行结果
JSResult.java	JS 结果对象，用于实现用户交互
JWebCoreJavaBridge.java	用 Java 与 WebCore 库中 Timer 和 Cookies 对象交互的桥接代码
LoadListener.java	载入器侦听器，用于处理载入器侦听消息
Network.java	该对象封装网络连接逻辑，为调用者提供更为高级的网络连接接口
PanZoom.java	用于处理图片缩放、移动等操作
PanZoomCellList.java	用于保存移动、缩放图片的 Cell
SslErrorHandler.java	用于处理 SSL 错误消息
StreamLoader.java	StreamLoader 抽象类是所有内容载入器对象的基类。该类是通过消息方式控制的状态机，用于将数据载入到 Frame 中
TextDialog.java	用于处理 HTML 中文本区域叠加情况，可以使用标准的文本编辑定义特殊 EditText 控件
URLUtil.java	URL 处理功能函数，用于编码、解码 URL 字符串，以及提供附加的 URL 类型分析功能
WebBackForwardList.java	该对象包含 WebView 对象中显示的历史数据
WebBackForwardListClient.java	浏览历史处理的客户接口类，所有需要接收浏览历史改变的类都需要实现该接口
WebChromeClient.java	Chrome 客户基类，Chrome 客户对象在浏览器文档标题、进度条、图标改变时会得到通知
WebHistoryItem.java	该对象用于保存一条网页历史数据
WebIconDataBase.java	图表数据库管理对象，所有的 WebView 均请求相同的图标数据库对象
WebSettings.java	WebView 的管理设置数据，该对象数据是通过 JNI 接口从底层获取
WebSyncManager.java	数据同步对象，用于 RAM 数据和 Flash 数据的同步操作
WebView.java	Web 视图对象，用于基本的网页数据载入、显示等 UI 操作
WebViewClient.java	Web 视图客户对象，在 Web 视图中有事件产生时，该对象可以获得通知
WebViewCore.java	该对象对 WebCore 库进行了封装，将 UI 线程中的数据请求发送给 WebCore 处理，并且通过 CallbackProxy 的方式，通过消息通知 UI 线程数据处理的结果
WebViewDatabase.java	该对象使用 SQLiteDatabase 为 WebCore 模块提供数据存取操作

下面将对 WebKit 模块的 Java 层的具体知识进行详细介绍。

16.3　Java 层的主要类

WebKit 模块的 Java 层一共由 41 个文件组成，本节将详细讲解各个主要类的基本信息，为读者学习本书后面的知识打下基础。

16.3.1　WebView 简介

WebView 类是 WebKit 模块 Java 层的视图类，所有需要使用 Web 浏览功能的 Android 应用程序都

要创建该视图对象显示和处理请求的网络资源。目前，WebKit 模块支持 HTTP、HTTPS、FTP 以及 JavaScript 请求。WebView 作为应用程序的 UI 接口，为用户提供了一系列的网页浏览、用户交互接口，客户程序通过这些接口访问 WebKit 核心代码。

在文件 WebView.java 中，WebView 类的主要实现代码如下所示。

```
public class WebView extends AbsoluteLayout
        implements ViewTreeObserver.OnGlobalFocusChangeListener,
        ViewGroup.OnHierarchyChangeListener, ViewDebug.HierarchyHandler {

    private static final String LOGTAG = "webview_proxy";

    private static Boolean sEnforceThreadChecking = false;
    public class WebViewTransport {
        private WebView mWebview;
        public synchronized void setWebView(WebView webview) {
            mWebview = webview;
        }
        public synchronized WebView getWebView() {
            return mWebview;
        }
    }
    public static final String SCHEME_TEL = "tel:";
    public static final String SCHEME_MAILTO = "mailto:";
    public static final String SCHEME_GEO = "geo:0,0?q=";
…
```

注意： WebView 类是一个非常重要的类，能够实现和网络有关的很多功能。在本章后面的内容中，将专门用一节的内容进行详细剖析。

16.3.2 WebViewDatabase

WebViewDatabase 类是 WebKit 模块中针对 SQLiteDatabase 对象的封装，用于存储和获取运行时浏览器保存的缓冲数据、历史访问数据、浏览器配置数据等。该对象是一个单实例对象，通过 getInstance()方法获取 WebViewDatabase 的实例。WebViewDatabase 是 WebKit 模块中的内部对象，仅供 WebKit 框架内部使用。

16.3.3 WebViewCore

WebViewCore 类是 Java 层与 C 层 WebKit 核心库的交互类，客户程序调用 WebView 的网页浏览相关操作会转发给 BrowserFrame 对象。当 WebKit 核心库完成实际的数据分析和处理后会回调 WebViweCore 中定义的一系列 JNI 接口，这些接口会通过 CallbackProxy 将相关事件通知相应的 UI 对象。

16.3.4 CallbackProxy

CallbackProxy 类是一个代理类，用于实现 UI 线程和 WebCore 线程之间的交互。CallbackProxy 类定义了一系列与用户相关的通知方法，当 WebCore 完成相应的数据处理后会调用 CallbackProxy 类中对应的方法，这些方法通过消息方式间接调用相应处理对象的处理方法。

16.3.5　BrowserFrame

BrowserFrame 类负责 URL 资源的载入、访问历史的维护、数据缓存等操作，该类会通过 JNI 接口直接与 WebKit C 层库交互。

16.3.6　JWebCoreJavaBridge

JWebCoreJavaBridge 类为 Java 层 WebKit 代码提供与 C 层 WebKit 核心部分的 Timer 和 Cookies 操作相关的方法。

16.3.7　DownloadManagerCore

DownloadManagerCore 类是一个下载管理核心类，主要负责管理网络资源的下载，所有的 Web 下载操作均由该类统一管理。该类实例运行在 WebKit 线程当中，与 UI 线程的交互是通过调用 CallbackProxy 对象中相应的方法完成的。

16.3.8　其他类

（1）WebSettings

WebSettings 类描述了 Web 浏览器访问相关的用户配置信息。

（2）DownloadListener

DownloadListener 类负责下载侦听接口，如果客户代码实现该接口，则在下载开始、失败、挂起、完成等情况下，DownloadManagerCore 对象会调用客户代码中实现的 DwonloadListener()方法。

（3）WebBackForwardList

WebBackForwarList 类负责维护用户访问的历史记录，该类为客户程序提供操作访问浏览器历史数据的相关方法。

（4）WebViewClient

在 WebViewClient 类中定义了一系列事件方法，如果 Android 应用程序设置了 WebViewClient 派生对象，则在页面载入、资源载入、页面访问错误等情况发生时，该派生对象的相应方法会被调用。

（5）WebBackForwardListClient

WebBackForwardListClient 类定义了对访问历史操作时可能产生的事件接口，当用户实现了该接口，则在操作访问历史时（访问历史移除、访问历史清空等），用户会得到通知。

（6）WebChromeClient

WebChromeClient 类定义了与浏览窗口修饰相关的事件。例如，接收到 Title、接收到 Icon、进度变化时，WebChromeClient 的相应方法会被调用。

16.4　数据载入器架构

在 WebKit 系统的 Java 部分框架中，使用数据载入器来加载相应类型的数据，目前有 CacheLoader、

DataLoader 以及 FileLoader 这 3 类载入器，分别用于处理缓存数据、内存数据以及文件数据的载入操作。Java 层（WebKit 模块）所有的载入器都从 StreamLoader 继承（其父类为 Handler），由于 StreamLoader 类的基类为 Handler 类，因此在构造载入器时，会开启一个事件处理线程，该线程负责实际的数据载入操作，而请求线程通过消息的方式驱动数据的载入。图 16-2 描述了数据载入器相关类的类图结构。

在 StreamLoader 类中定义了如下 4 个不同的消息。

- ☑ MSG_STATUS：表示发送状态消息。
- ☑ MSG_HEADERS：表示发送消息头消息。
- ☑ MSG_DATA：表示发送数据消息。
- ☑ MSG_END：表示数据发送完毕消息。

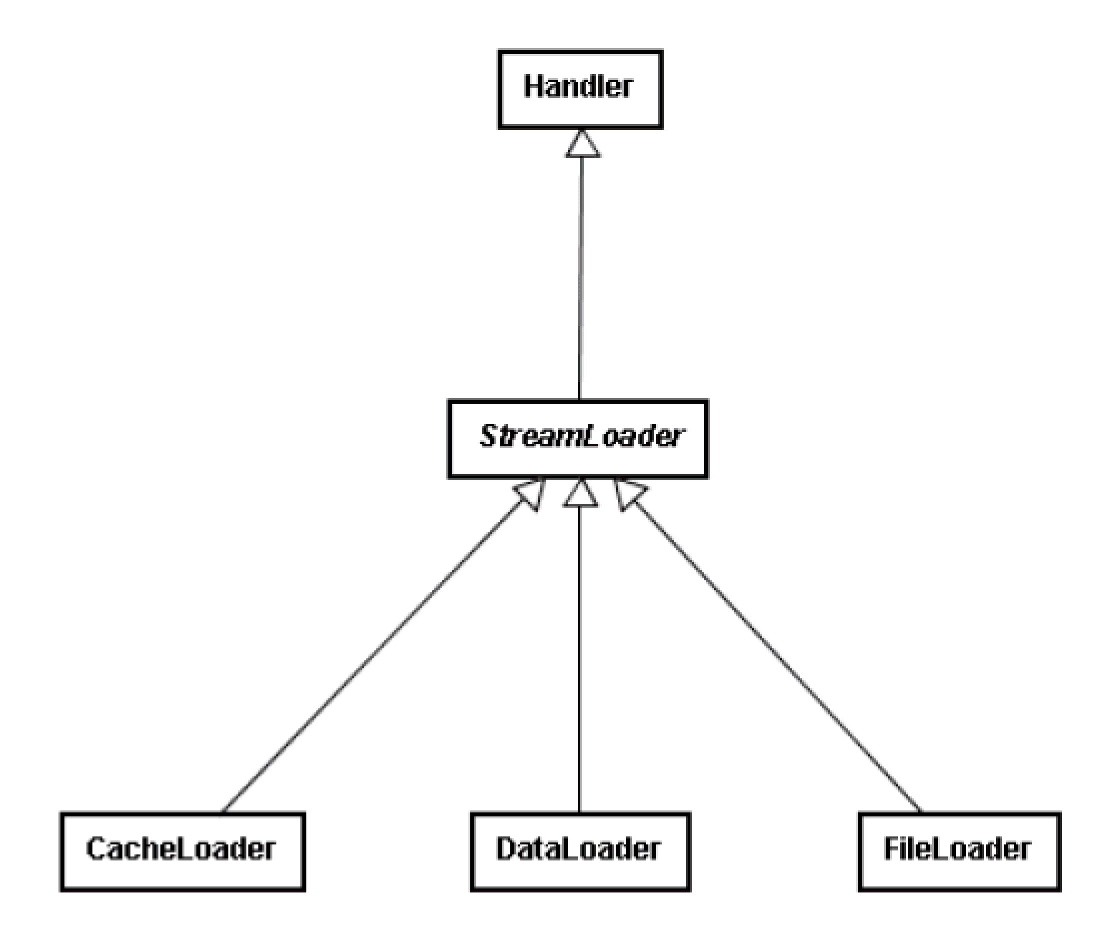


图 16-2　数据载入器的类图结构

在 StreamLoader 类中提供了两个抽象保护方法以及一个共有方法，其中，保护方法 setupStreamAndSendStatus() 用于构造与通信协议相关的数据流，以及向 LoadListener 发送状态。方法 buildHeaders()负责向子类提供构造特定协议消息头功能。所有载入器只有一个共有方法（load()），因此当需要载入数据时，只需调用该方法即可。与数据载入流程相关的类还有 LoaderListener 和 BrowserFrame，当发生数据载入事件时，WebKit 的 C 库会更新载入进度，并且会通知 BrowserFrame，BrowserFrame 接收到进度条变更事件后会通过 CallbackProxy 对象，通知 View 类进度条数据变更。

16.5　Java 层对应的 C/C++类库

每一个 Java 类在下面的 C/C++层都会有一个对应的类库，各个 Java 类和 C/C++类库对应关系的具体说明如表 16-2 所示。

表 16-2　Java 层中的类和 C/C++类库的对应关系

类	功 能 描 述
ChromeClientAndroid	该类主要处理 WebCore 中与 Frame 装饰相关的操作。例如，设置状态栏、滚动条、JavaScript 脚本提示框等。当浏览器中有相关事件产生，ChromeClientAndroid 类的相应方法会被调用，该类会将相关的 UI 事件通过 Bridge 传递给 Java 层，由 Java 层负责绘制以及用户交互方面的处理
EditorClientAndroid	该类负责处理页面中文本相关的处理，例如，文本输入、取消、输入法数据处理、文本粘贴、文本编辑等操作。不过目前该类只对按键相关的时间进行了处理，其他操作均未支持
ContextMenuClient	该类提供页面相关的功能菜单，例如，图片复制、朗读、查找等功能。但是，目前项目中未实现具体功能
DragClient	该类定义了与页面拖曳相关的处理，但是目前该类没有实现具体功能
FrameLoaderClientAndroid	该类提供与 Frame 加载相关的操作，当用户请求加载一个页面时，WebCore 分析完网页数据后，会通过该类调用 Java 层的回调方法，通知 UI 相关的组件处理

续表

类	功 能 描 述
InspectorClientAndroid	该类提供与窗口相关的操作，例如，窗口显示、关闭窗口、附加窗口等。不过目前该类的各个方法均为空实现
Page	该类提供与页面相关的操作，例如，网页页面的前进、后退等操作
FrameAndroid	该类为 Android 提供 Frame 管理
FrameBridge	该类对 Frame 相关的 Java 层方法进行了封装，当有 Frame 事件产生时，WebCore 通过 FrameBridge 回调 Java 的回调函数，完成用户交互过程
AssetManager	该类为浏览器提供本地资源访问功能
RenderSkinAndroid	该类与控件绘制相关，所有需绘制的控件都需要从该类派生，目前 WebKit 模块中有 Button、Combo 和 Radio 这 3 类控件

下面将详细讲解 WebKit 中 C/C++层各个库的基本知识。

（1）BrowserFrame

与 Java 类 BrowserFrame 相对应的 C++类为 FrameBridge，该类为 Dalvik 虚拟机回调 BrowserFrame 类中定义的本地方法进行了封装。与 BrowserFrame 中回调函数（Java 层）相对应的 C 层结构定义代码如下所示。

```
struct FrameBridge::JavaBrowserFrame
{
    JavaVM* mJVM;
    jobject mObj;
    jmethodID mStartLoadingResource;
    jmethodID mLoadStarted;
    jmethodID mUpdateHistoryForCommit;
    jmethodID mUpdateCurrentHistoryData;
    jmethodID mReportError;
    jmethodID setTitle;
    jmethodID mWindowObjectCleared;
    jmethodID mDidReceiveIcon;
    jmethodID mUpdateVisiteHistory;
    jmethodID mHandleUrl;
    jmethodID mCreateWindow;
    jmethodID mCloseWindow;
    jmethodID mDecidePolicyForFormResubmission;
};
```

在上述代码结构中，mJavaFrame 作为 FrameBridge（C 层）的一个成员变量，在 FrameBridge 构造函数中用类 BrowserFrame（Java 层）的回调方法的偏移量初始化 JavaBrowserFrame 结构的各个域。当初始工作完成后，当 WebCore（C 层）在剖析网页数据时，和 Frame 相关的资源会发生改变（例如 Web 页面的主题变化），此时会通过 mJavaFrame 结构调用指定 BrowserFrame 对象的相应方法，并通知 Java 层进行处理。

注意：为了节省本书的篇幅，后面各个类库的实现代码将不再一一列出。

（2）JWebCoreJavaBridge

与该对象相对应的 C 层对象为 JavaBridge，JavaBridge 对象继承了 TimerClient 和 CookieClient 类，

负责 WebCore 中的定时器和 Cookie 管理。与 Java 层 JWebCoreJavaBridge 类中方法偏移量相关的是 JavaBridege 中的几个成员变量，在构造 JavaBridge 对象时，会初始化这些成员变量，之后有 Timer 或者 Cookies 事件产生，WebCore 会通过这些 ID 值回调对应 JWebCoreJavaBridge 的相应方法。

（3）LoadListener

与该对象相关的 C 层结构是 struct resourceloader_t，该结构保存了 LoadListener 对象 ID、CancelMethod ID 以及 DownloadFiledMethod ID 值。当有 Cancel 或者 Download 事件产生，WebCore 会回调 LoadListener 类中的 CancelMethod 或者 DownloadFileMethod。

（4）WebViewCore

与 WebViewCore 相关的 C 类是 WebCoreViewImpl，WebViewCoreImpl 类有一个 JavaGlue 对象作为成员变量，在构建 WebCoreViewImpl 对象时，用 WebViewCore（Java 层）中的方法 ID 值初始化该成员变量，并且会将构建的 WebCoreViewImpl 对象指针复制给 WebViewCore（Java 层）的 mNativeClass，这样将 WebViewCore（Java 层）和 WebViewCoreImple（C 层）关联起来。

（5）WebSettings

与 WebSettings 相关的 C 层结构是 struct FieldIds，该结构保存了 WebSettings 类中定义的属性 ID 以及方法 ID，在 WebCore 初始化时（WebViewCore 的静态方法中使用 System.loadLibrary 载入）会设置这些方法和属性的 ID 值。

（6）WebView

与 WebView 相关的 C 层类是 WebViewNative，在该类的 mJavaGlue 中保存着 WebView 中定义的属性和方法 ID，在 WebViewNative 构造方法中初始化，并且将构造的 WebViewNative 对象的指针赋值给 WebView 类的 mNativeClass 变量，这样 WebView 和 WebViewNative 对象建立了关系。

16.6 分析 WebKit 的操作过程

经过本章前面内容的学习，已经基本了解了 WebKit 系统中各层主要类的功能。下面将简单介绍和 WebKit 相关的基本操作知识，为读者学习本书后面的知识打下基础。

16.6.1 WebKit 初始化

在 Android SDK 中提供了 WebView 类，使用此类可以提供客户化浏览显示功能。如果客户需要加入浏览器的支持，可将该类的实例或者派生类的实例作为视图，调用 Activity 类的 setContentView 显示给用户。当客户代码中第一次生成 WebView 对象时，会初始化 WebKit 库（包括 Java 层和 C 层两个部分），之后用户可以操作 WebView 对象完成网络或者本地资源的访问。

WebView 对象的生成主要涉及 3 个类：CallbackProxy、WebViewCore 以及 WebViewDatabase。其中，CallbackProxy 对象为 WebKit 模块中 UI 线程和 WebKit 类库提供交互功能，WebViewCore 是 WebKit 的核心层，负责与 C 层交互以及 WebKit 模块 C 层类库初始化，而 WebViewDatabase 为 WebKit 模块运行时缓存、数据存储提供支持。

初始化的过程就是使用 WebView 创建 CallbackProxy 对象和 WebViewCore 对象的过程。WebKit 模块初始化流程如下所示。



（1）调用 System.loadLibrary 载入 webcore 相关类库（C 层）。

（2）如果是第一次初始化 WebViewCore 对象，创建 WebCoreTherad 线程。

（3）创建 EventHub 对象，处理 WebViewCore 事件。

（4）获取 WebIconDatabase 对象实例。

（5）向 WebCoreThread 发送初始化消息。

根据上述流程，假如要获取 WebViewDatabase 实例，则可以按照下面的步骤实现。

（1）调用 System.loadLibrary()方法载入 webcore 相关类库，该过程由 Dalvik 虚拟机完成，它会从动态链接库目录中寻找 libWebCore.so 类库载入内存中，并且调用 WebKit 初始化模块的 JNI_OnLoad()方法。WebKit 模块的 JNI_OnLoad 方法中完成了如下初始化操作。

- ☑ 初始化 framebridge[register_android_webcore_framebridge]：初始化 gFrameAndroidField 静态变量以及注册 BrowserFrame 类中的本地方法表。
- ☑ 初始化 javabridge[register_android_webcore_javabridge]：初始化 gJavaBridge.mObject 对象，以及注册 JWebCoreJavaBridge 类中的本地方法。
- ☑ 初始化资源 loader[register_android_webcore_resource_loader]：初始化 gResourceLoader 静态变量，以及注册 LoadListener 类的本地方法。
- ☑ 初始化 webviewcore[register_android_webkit_webviewcore]：初始化 gWebCoreViewImplField 静态变量，以及注册 WebViewCore 类的本地方法。
- ☑ 初始化 webhistory[register_android_webkit_webhistory]：初始化 gWebHistoryItem 结构，以及注册 WebBackForwardList 和 WebHistoryItem 类的本地方法。
- ☑ 初始化 webicondatabase[register_android_webkit_webicondatabase]：注册 WebIconDatabase 类的本地方法。
- ☑ 初始化 websettings[register_android_webkit_websettings]：初始化 gFieldIds 静态变量，以及注册 WebSettings 类的本地方法。
- ☑ 初始化 webview[register_android_webkit_webview]：初始化 gWebViewNativeField 静态变量，以及注册 WebView 类的本地方法。

（2）实现 WebCoreThread 初始化，该初始化只在第一次创建 WebViewCore 对象时完成，当用户代码第一次生成 WebView 对象，会在初始化 WebViewCore 类时创建 WebCoreThread 线程，该线程负责处理 WebCore 初始化事件。此时 WebViewCore 构造函数会被阻塞，直到一个 WebView 初始化请求完毕时，会在 WebCoreThread 线程中唤醒。

（3）创建 EventStub 对象，该对象处理 WebView 类的事件，当 WebCore 初始化完成后会向 WebView 对象发送事件，WebView 类的 EventStub 对象处理该事件，并且完成后续初始化工作。

（4）获取 WebIconDatabase 对象实例。

（5）向 WebViewCore 发送 INITIALIZE 事件，并且将 this 指针作为消息内容传递。WebView 类主要负责处理 UI 相关的事件，而 WebViewCore 主要负责与 WebCore 库交互。在运行时期，UI 线程和 WebCore 数据处理线程运行在两个独立的线程当中。WebCoreThread 线程接收到 INITIALIZE 线程后，会调用消息对象参数的 initialize()方法，而后唤醒阻塞的 WebViewCore Java 线程（该线程在 WebViewCore 的构造函数中被阻塞）。不同的 WebView 对象实例有不同的 WebViewCore 对象实例，因此通过消息的方式可以使得 UI 线程和 WebViewCore 线程解耦合。WebCoreThread 的事件处理函数处理 INITIALIZE 消息时，调用的是不同 WebView 中 WebViewCore 实例的 initialize()方法。

WebViewCore 类中的 initialize()方法中会创建 BrowserFrame 对象（该对象管理整个 Web 窗体，以及 Frame 相关事件），并且向 WebView 对象发送 WEBCORE_INITIALIZED_MSG_ID 消息。WebView 消息处理函数能够根据其参数来初始化指定 WebViewCore 对象，并且能够更新 WebViewCore 的 Frame 缓冲。

16.6.2 载入数据

（1）载入网络数据

在 Android 应用开发过程中，可以使用类 WebView 的 loadUrl()方法请求访问指定的 URL 网页数据。在 WebView 对象中保存着 WebViewCore 的引用，由于 WebView 属于 UI 线程，而 WebViewCore 属于后台线程，因此 WebView 对象的 loadUrl 被调用时，会通过消息的方式将 URL 信息传递给 WebViewCore 对象，该对象会调用成员变量 mBrowserFrame 的 loadUrl()方法，进而调用 WebKit 库完成数据的载入。

当载入网络数据时，此功能分别由 Java 层和 C 层共同完成，其中，Java 层负责完成用户交互、资源下载等操作，而 C 层主要完成数据分析（建立 DOM 树、分析页面元素等）操作。由于 UI 线程和 WebCore 线程运行在不同的两个线程中，因此当用户请求访问网络资源时，通过消息的方式向 WebViewCore 对象发送载入资源请求。

在 Java 层的 WebKit 模块中，所有与资源载入相关的操作都是由 BrowserFrame 类中对应的方法完成的，这些方法是本地方法，会直接调用 WebCore 库的 C 层函数完成数据载入请求，以及资源分析等操作。C 层的 FrameLoader 类是浏览框架的资源载入器，该类负责检查访问策略以及向 Java 层发送下载资源请求等功能。在 FrameLoader 中，当用户请求网络资源时，经过一系列的策略检查后会调用 FrameBridge 的 startLoadingResource()方法，该方法会回调 BrowserFrame（Java）类的 startLoadingResource()方法，完成网络数据的下载，然后类 BrowserFrame（Java）的方法 startLoadingResource()会返回一个 LoadListener 的对象，FrameLoader 会删除原有的 FrameLoader 对象，将 LoadListener 对象封装成 ResourceLoadHandler 对象，并且将其设置为新的 FrameLoader。到此完成了一次资源访问请求，接下来库 WebCore 会根据资源数据进行分析和构建 DOM，以及构建相关的数据结构。

（2）载入本地数据

所谓本地数据是指以 data://开头的 URL，载入本地数据的过程和载入网络数据的方法一样，只不过在执行 FrameLoader 类的 executeLoad()方法时，会根据 URL 的 SCHEME 类型区分，调用 DataLoader 的 requestUrl()方法，而不是调用 handleHTTPLoad 建立实际的网络通信连接。

（3）载入文件数据

所谓文件数据是指以 file://开头的 URL，载入的基本流程与网络数据载入流程基本一致，不同的是在运行 FrameLoader 类的 executeLoad()方法时，根据 SCHEME 类型，调用 FileLoader 的 requestUrl()方法来完成数据加载。

16.6.3 刷新绘制

当用户拖动滚动条、有窗口遮盖或者有页面事件触发，都会向 WebViewCore（Java 层）对象发送背景重绘消息，该消息会引起网页数据的绘制操作。WebKit 的数据绘制可能出于效率上的考虑，没有



通过 Java 层，而是直接在 C 层使用 SGL 库完成。与 Java 层图形绘制相关的 Java 对象有 3 个，具体说明如下所示。

（1）Picture 类

该类对 SGL 封装，其中，变量 mNativePicture 实际上是保存着 SkPicture 对象的指针。WebViewCore 中定义了两个 Picture 对象，当作双缓冲处理，在调用 webKitDraw()方法时，会交换两个缓冲区，加速刷新速度。

（2）WebView 类

该类接受用户交互相关的操作，当有滚屏、窗口遮盖、用户点击页面按钮等相关操作时，WebView 对象会向与之相关的 WebViewCore 对象发送 VIEW_SIZE_CHANGED 消息。当 WebViewCore 对象接收到该消息后，将构建时建立的 mContentPictureB 刷新到屏幕上，然后将 mContentPictureA 与之交换。

（3）WebViewCore 类

该类封装了 WebKit 的 C 层代码，为视图类提供对 WebKit 的操作接口，所有对 WebKit 库的用户请求均由该类处理，并且该类还为视图类提供了两个 Picture 对象，用于图形数据刷新。

例如，在拖曳 Web 页面时，当用户点击触摸屏并且移动手指时会引发 touch 事件，Android 平台会将 touch 事件传递给最前端的视图响应（dispatchTouchEvent()方法处理）。在 WebView 类中定义了 5 种 touchJ 模式，在手指拖动 Web 页面的情况下，会触发 mMotionDragMode，并且会调用 View 类的 scrollBy()方法，触发滚屏事件以及使视图无效（重绘，会调用 View 的 onDraw()方法）。WebView 视图中的滚屏事件由 onScrollChanged()方法响应，该方法向 WebViewCore 对象发送 SET_VISIBLE_RECT 事件。

WebViewCore 对象接收到 SET_VISIBLE_RECT 事件后，将消息参数中保存的新视图的矩形区域大小传递给 nativeSetVisibleRect()方法，通知 WebCoreViewImpl 对象（C 层）视图矩形变更（WebCoreViewImpl::setVisibleRect 方法）。在 setVisibleRect()方法中，会通过虚拟机调用 WebViewCore 的 contentInvalidate()方法，该方法会引发 webkitDraw()方法的调用（通过 WEBKIT_DRAW 消息）。在方法 webkitDraw()中，首先会将 mContentPictureB 对象传递给本地方法 nativeDraw()绘制，然后将 mContentPictureB 的内容与 mContentPictureA 的内容互换。此处 mContentPictureA 缓冲区是供 WebViewCore 的 draw()方法使用，如果用户选择某个控件，绘制焦点框时 WebViewCore 对象的 draw()方法会调用，绘制的内容保存在 mContentPictureA 中，之后会通过 Canvas 对象（Java 层）的 drawPicture()方法将其绘制到屏幕上，而 mContentPictureB 缓冲区是用于 built 操作的，nativeDraw()方法中首先会将传递的 mContentPictureB 对象数据重置，而后在重新构建的 mContentPictureB 画布上，将层上相关的元素绘制到该画布上。然后将 mContentPictureB 和 mContentPictureA 的内容互换，这样一次重绘事件产生时（会调用 WebView.onDraw()方法）会将 mContentPictureA 的数据使用 Canvas()类的 drawPicture()绘制到屏幕上。当 webkitDraw()方法将 mContentPictureA 与 mContentPictureB 指针对调后，会向 WebView()对象发送 NEW_PICTURE_MSG_ID 消息，该消息会引发 WebViewCore 的 VIEW_SIZE_CHANGED 消息的产生，并且会使当前视图无效产生重绘事件（invalidate()），引发 onDraw()方法的调用，完成一次网页数据的绘制过程。

16.7　WebViewCore 详解

在 Android 5.0 系统中，文件 WebViewCore.java 位于目录 Frameworks/base/core/java/android/webkit 中。

WebViewCore 类是 Java 层与 C 层 WebKit 核心库的交互类，客户程序调用 WebView 的网页浏览相关操作会转发给 BrowserFrame 对象。当 WebKit 核心库完成实际的数据分析和处理后会回调 WebViweCore 中定义的一系列 JNI 接口，这些接口会通过 CallbackProxy 将相关事件通知相应的 UI 对象。文件 WebViewCore.java 的主要实现代码如下所示。

```
private void initialize() {
    /* Initialize our private BrowserFrame class to handle all
     * frame-related functions. We need to create a new view which
     * in turn creates a C level FrameView and attaches it to the frame
     */
    mBrowserFrame = new BrowserFrame(mContext, this, mCallbackProxy,
            mSettings, mJavascriptInterfaces);
    mJavascriptInterfaces = null;
    mSettings.syncSettingsAndCreateHandler(mBrowserFrame);
    WebIconDatabaseClassic.getInstance().createHandler();
    WebStorageClassic.getInstance().createHandler();
    GeolocationPermissionsClassic.getInstance().createHandler();
    mEventHub.transferMessages();

    if (mWebViewClassic != null) {
        Message.obtain(mWebViewClassic.mPrivateHandler,
                WebViewClassic.WEBCORE_INITIALIZED_MSG_ID,
                mNativeClass, 0).sendToTarget();
    }
}
private int mapDirection(int webkitDirection) {
    /*
     * This is WebKit's FocusDirection enum (from FocusDirection.h)
    enum FocusDirection {
        FocusDirectionNone = 0,
        FocusDirectionForward,
        FocusDirectionBackward,
        FocusDirectionUp,
        FocusDirectionDown,
        FocusDirectionLeft,
        FocusDirectionRight
    };
     */
    switch (webkitDirection) {
    case 1:
        return View.FOCUS_FORWARD;
    case 2:
        return View.FOCUS_BACKWARD;
    case 3:
        return View.FOCUS_UP;
    case 4:
        return View.FOCUS_DOWN;
    case 5:
        return View.FOCUS_LEFT;
    case 6:
```

```
            return View.FOCUS_RIGHT;
        }
        return 0;
    }
    private String openFileChooser(String acceptType, String capture) {
        Uri uri = mCallbackProxy.openFileChooser(acceptType, capture);
        if (uri != null) {
            String filePath = "";
            Cursor cursor = mContext.getContentResolver().query(
                    uri,
                    new String[] { MediaStore.Images.Media.DATA },
                    null, null, null);
            if (cursor != null) {
                try {
                    if (cursor.moveToNext()) {
                        filePath = cursor.getString(0);
                    }
                } finally {
                    cursor.close();
                }
            } else {
                filePath = uri.getLastPathSegment();
            }
            String uriString = uri.toString();
            BrowserFrame.sJavaBridge.storeFilePathForContentUri(filePath, uriString);
            return uriString;
        }
        return "";
    }
    protected void populateVisitedLinks() {
        ValueCallback callback = new ValueCallback<String[]>() {
            @Override
            public void onReceiveValue(String[] value) {
                sendMessage(EventHub.POPULATE_VISITED_LINKS, (Object)value);
            }
        };
        mCallbackProxy.getVisitedHistory(callback);
    }
    private static class WebCoreThread implements Runnable {
        private static final int INITIALIZE = 0;
        private static final int REDUCE_PRIORITY = 1;
        private static final int RESUME_PRIORITY = 2;

        @Override
        public void run() {
            Looper.prepare();
            Assert.assertNull(sWebCoreHandler);
            synchronized (WebViewCore.class) {
                sWebCoreHandler = new Handler() {
                    @Override
                    public void handleMessage(Message msg) {
```

```
            switch (msg.what) {
                case INITIALIZE:
                    WebViewCore core = (WebViewCore) msg.obj;
                    core.initialize();
                    break;

                case REDUCE_PRIORITY:
                    Process.setThreadPriority(
                            Process.THREAD_PRIORITY_DEFAULT + 3 *
                            Process.THREAD_PRIORITY_LESS_FAVORABLE);
                    break;

                case RESUME_PRIORITY:
                    Process.setThreadPriority(
                            Process.THREAD_PRIORITY_DEFAULT);
                    break;

                case EventHub.ADD_PACKAGE_NAME:
                    if (BrowserFrame.sJavaBridge == null) {
                        throw new IllegalStateException(
                                "No WebView has been created in this process!");
                    }
                    BrowserFrame.sJavaBridge.addPackageName((String) msg.obj);
                    break;

                case EventHub.REMOVE_PACKAGE_NAME:
                    if (BrowserFrame.sJavaBridge == null) {
                        throw new IllegalStateException(
                                "No WebView has been created in this process!");
                    }
                    BrowserFrame.sJavaBridge.removePackageName((String) msg.obj);
                    break;

                case EventHub.PROXY_CHANGED:
                    if (BrowserFrame.sJavaBridge == null) {
                        throw new IllegalStateException(
                                "No WebView has been created in this process!");
                    }
                    BrowserFrame.sJavaBridge.updateProxy((ProxyProperties)msg.obj);
                    break;

                case EventHub.HEARTBEAT:
                    Message m = (Message)msg.obj;
                    m.sendToTarget();
                    break;
                case EventHub.TRUST_STORAGE_UPDATED:
                    nativeCertTrustChanged();
                    CertificateChainValidator.handleTrustStorageUpdate();
                    break;
            }
        }
```

```
                };
                WebViewCore.class.notify();
            }
            Looper.loop();
        }
}
        static final String[] HandlerDebugString = {
            "REVEAL_SELECTION", // 96
            "", // 97
            "", // = 98
            "SCROLL_TEXT_INPUT", // = 99
            "LOAD_URL", // = 100;
            "STOP_LOADING", // = 101;
            "RELOAD", // = 102;
            "KEY_DOWN", // = 103;
            "KEY_UP", // = 104;
            "VIEW_SIZE_CHANGED", // = 105;
            "GO_BACK_FORWARD", // = 106;
            "SET_SCROLL_OFFSET", // = 107;
            "RESTORE_STATE", // = 108;
            "PAUSE_TIMERS", // = 109;
            "RESUME_TIMERS", // = 110;
            "CLEAR_CACHE", // = 111;
            "CLEAR_HISTORY", // = 112;
            "SET_SELECTION", // = 113;
            "REPLACE_TEXT", // = 114;
            "PASS_TO_JS", // = 115;
            "SET_GLOBAL_BOUNDS", // = 116;
            "", // = 117;
            "CLICK", // = 118;
            "SET_NETWORK_STATE", // = 119;
            "DOC_HAS_IMAGES", // = 120;
            "FAKE_CLICK", // = 121;
            "DELETE_SELECTION", // = 122;
            "LISTBOX_CHOICES", // = 123;
            "SINGLE_LISTBOX_CHOICE", // = 124;
            "MESSAGE_RELAY", // = 125;
            "SET_BACKGROUND_COLOR", // = 126;
            "SET_MOVE_FOCUS", // = 127
            "SAVE_DOCUMENT_STATE", // = 128;
            "129", // = 129;
            "WEBKIT_DRAW", // = 130;
            "131", // = 131;
            "POST_URL", // = 132;
            "", // = 133;
            "CLEAR_CONTENT", // = 134;
            "", // = 135;
            "", // = 136;
            "REQUEST_CURSOR_HREF", // = 137;
            "ADD_JS_INTERFACE", // = 138;
            "LOAD_DATA", // = 139;
```

```
            "", // = 140;
            "", // = 141;
            "SET_ACTIVE", // = 142;
            "ON_PAUSE",      // = 143
            "ON_RESUME",     // = 144
            "FREE_MEMORY",   // = 145
            "VALID_NODE_BOUNDS", // = 146
            "SAVE_WEBARCHIVE", // = 147
            "WEBKIT_DRAW_LAYERS", // = 148;
            "REMOVE_JS_INTERFACE", // = 149;
    };
```

与文件 WebViewCore.java 相关的 C 类是 WebViewCoreI，在此类中定义了两个数据结构，一个是 WebViewCoreFields，对应于 Java 层 WebViewCore 对象的成员变量；另一个是 WebViewCore::JavaGlue，对应于 Java 层 WebViewCore 对象的成员方法。具体的定义代码如下所示。

```
struct WebViewCoreFields {
    jfieldID m_nativeClass;
    jfieldID m_viewportWidth;
    jfieldID m_viewportHeight;
    jfieldID m_viewportInitialScale;
    jfieldID m_viewportMinimumScale;
    jfieldID m_viewportMaximumScale;
    jfieldID m_viewportUserScalable;
    jfieldID m_viewportDensityDpi;
    jfieldID m_webView;
    jfieldID m_drawIsPaused;
    jfieldID m_lowMemoryUsageMb;
    jfieldID m_highMemoryUsageMb;
    jfieldID m_highUsageDeltaMb;
} gWebViewCoreFields;

// ——————————————————————————————————————-

struct WebViewCore::JavaGlue {
    jweak m_obj;
    jmethodID m_scrollTo;
    jmethodID m_contentDraw;
    jmethodID m_layersDraw;
    jmethodID m_requestListBox;
    jmethodID m_openFileChooser;
    jmethodID m_requestSingleListBox;
    jmethodID m_jsAlert;
    jmethodID m_jsConfirm;
    jmethodID m_jsPrompt;
    jmethodID m_jsUnload;
    jmethodID m_jsInterrupt;
    jmethodID m_didFirstLayout;
    jmethodID m_updateViewport;
    jmethodID m_sendNotifyProgressFinished;
    jmethodID m_sendViewInvalidate;
```

```
    jmethodID m_updateTextfield;
    jmethodID m_updateTextSelection;
    jmethodID m_clearTextEntry;
    jmethodID m_restoreScale;
    jmethodID m_needTouchEvents;
    jmethodID m_requestKeyboard;
    jmethodID m_requestKeyboardWithSelection;
    jmethodID m_exceededDatabaseQuota;
    jmethodID m_reachedMaxAppCacheSize;
    jmethodID m_populateVisitedLinks;
    jmethodID m_geolocationPermissionsShowPrompt;
    jmethodID m_geolocationPermissionsHidePrompt;
    jmethodID m_getDeviceMotionService;
    jmethodID m_getDeviceOrientationService;
    jmethodID m_addMessageToConsole;
    jmethodID m_formDidBlur;
    jmethodID m_getPluginClass;
    jmethodID m_showFullScreenPlugin;
    jmethodID m_hideFullScreenPlugin;
    jmethodID m_createSurface;
    jmethodID m_addSurface;
    jmethodID m_updateSurface;
    jmethodID m_destroySurface;
    jmethodID m_getContext;
    jmethodID m_keepScreenOn;
    jmethodID m_sendFindAgain;
    jmethodID m_showRect;
    jmethodID m_centerFitRect;
    jmethodID m_setScrollbarModes;
    jmethodID m_setInstallableWebApp;
    jmethodID m_enterFullscreenForVideoLayer;
    jmethodID m_setWebTextViewAutoFillable;
    jmethodID m_selectAt;
    AutoJObject object(JNIEnv* env) {
        return getRealObject(env, m_obj);
    }
};
```

在类 WebViewCore 中有一个作为成员变量的对象——JavaGlue，在构建 WebViewCore 对象时，用 WebViewCore（Java 层）中的方法 ID 值初始化该成员变量，并且会将构建的 WebViewCore 对象指针复制给 WebViewCore（Java 层）的 mNativeClass，这样可将 WebViewCore（Java 层）和 WebViewCore（C 层）关联起来。

第 17 章　Android 5.0 中的 WebView

文件 WebView.java 实现了类 WebView，此类是 WebKit 模块 Java 层的视图类，所有需要使用 Web 浏览功能的 Android 应用程序都要创建该视图对象显示和处理请求的网络资源。目前，WebKit 模块支持 HTTP、HTTPS、FTP 以及 JavaScript 请求。从 Android 5.0 开始，引入了 Mixed Content（图文内容混合）模式和第三方 Cookie。本章将详细讲解 Android 5.0 系统中 WebView 的核心架构知识，为读者深入理解 Android 网络系统的架构打下基础。

17.1　WebView 架构基础

在 Android 5.0 系统中，文件 WebView.java 是一个内置的支持浏览器的视图 View，此文件位于目录 Frameworks/base/core/java/android/webkit 中。

文件 WebView.java 实现了类 WebView，此类是 WebKit 模块 Java 层的视图类，所有需要使用 Web 浏览功能的 Android 应用程序都要创建该视图对象显示和处理请求的网络资源。目前，WebKit 模块支持 HTTP、HTTPS、FTP 以及 JavaScript 请求。WebView 作为应用程序的 UI 接口，为用户提供了一系列的网页浏览、用户交互接口，客户程序通过这些接口访问 WebKit 核心代码。可以说 WebView 实现了 WebKit 的最核心功能。近年来随着用户对网络安全的重视，谷歌公司为了创建一个安全、稳定和快速的通用浏览器系统，特意推出了 Chromium 引擎驱动，这也是当前谷歌官方浏览器 Chrome 的引擎驱动。在全新的 Android 5.0 系统中，WebView 将开始采用 Chromium 引擎驱动。

浏览 Android 5.0 源码时会发现，之前版本中的 external/WebKit 目录已经被移除，取而代之的是 chromium_org。由此可见，Chromium 已经完全取代了之前的 WebKit for Android。虽然如此，但是 Android WebView 的 API 接口并没有变，与旧版本完全兼容。这样的好处是基于 WebView 构建的 APP，无须做任何修改即可享受 Chromium 内核的高效与强大。

在 Android 5.0 系统中，frameworks/base/core/java/android/webkit 目录下的各个文件如图 17-1 所示。

在 Android 5.0 中，设计者抽象出了一个 WebViewProvider 接口，WebView 本身并不实现具体的功能，而是将所有的处理功能交给 WebViewProvider 来实现。而 WebViewProvider 只是一个接口，具体采用哪一种引擎交给实现者来决定。其实从 Android 4.1 系统开始就采用了这种架构模式，这说明谷歌那时便已经开始为以 Chromium 为驱动的 WebView 作准备。在 Android 5.0 之前的版本，通过 WebViewClassic 实现 WebViewProvider 接口。而在 Android 5.0 系统中，则通过 WebViewChromium 实现 WebViewProvider 接口。

WebViewFactory 也采用了相似的结构，设置了实例化 WebViewFactoryProvider 的具体策略。在 WebViewFactoryProvider 中有一个十分关键的接口 createWebView，功能是创建具体的 WebViewProvider。

Android 5.0 中 WebView 的代码结构如图 17-2 所示。

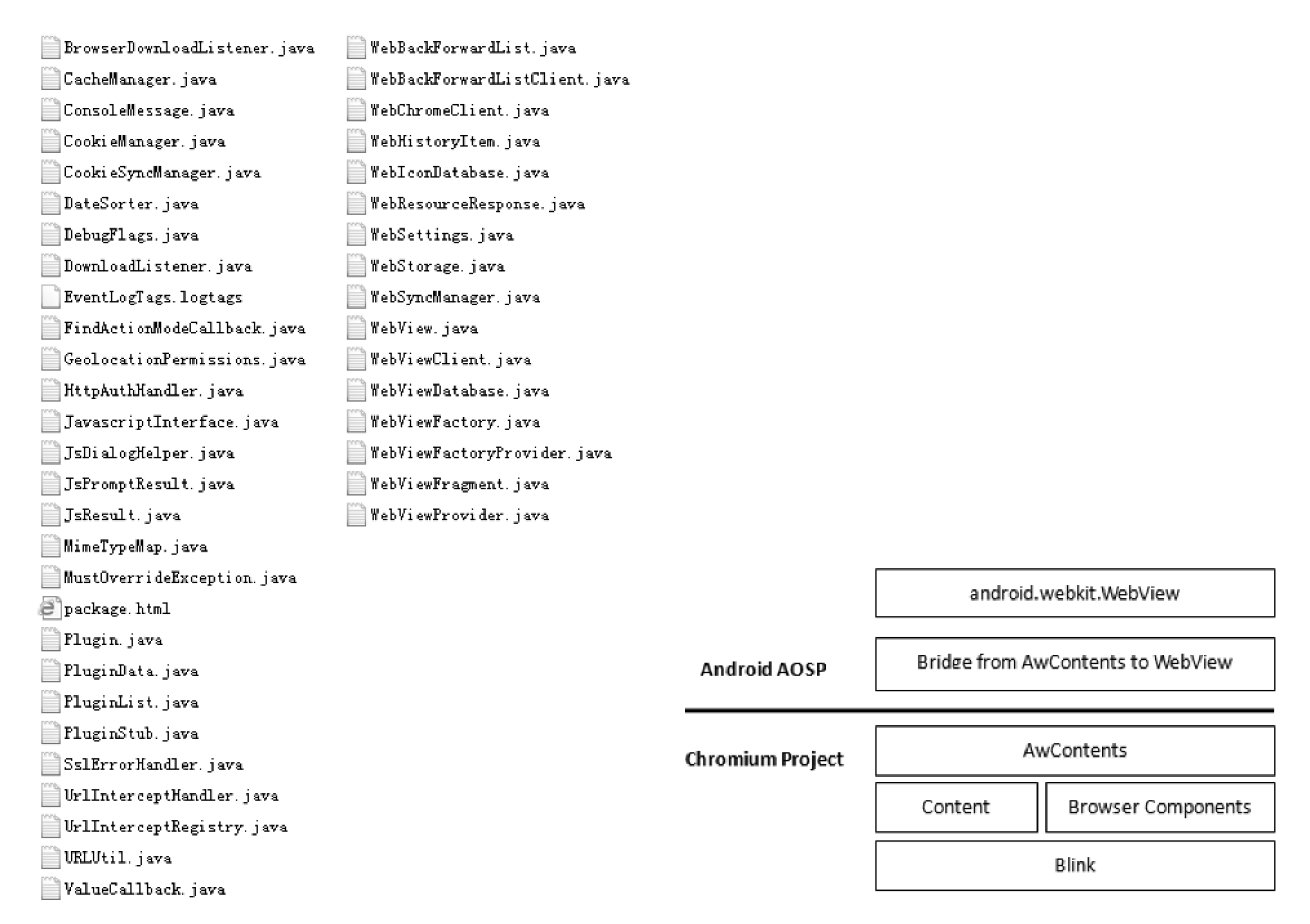


图 17-1　frameworks/base/core/java/android/webkit 目录下的文件　　图 17-2　Android 5.0 中 WebView 的代码结构

由此可见，传统的基于 AOSP master 架构在 WebKit 实现中仍然保留，因为采用了灵活的架构，所以可以容易地在 AOSP 和 Chromium 两种核心之间实现切换。

在 ContentAPI 之上，Chromium 的 WebView 实现封装了一个新的类 AwContents，该类主要基于 ContentViewCore 类的实现，不同的是，AwContents 需要基于一个原来存在于 chrome/目录下的模块（图 17-2 中的 Browser Components），但是 AwContents 不应该依赖该目录，所以，将 chrome 中的一些所谓的浏览器模块化是 Chromium 的一个方向。目前，一些模块已经从 chrome 中抽取出来了，具体请浏览 components/。

因为在 AwContents 中提供的不是 WebView 的 API，所以需要借助中间桥接层，将 AwContents 桥接到 WebView，这就是图 17-2 中的桥接模块，该模块位于 Android 5.0 源代码中的 frameworks/webview/chromium/java/com/android/webview/chromium/目录下。WebViewChromium 类和 WebViewChromiumFactory 类作为 WebView 的具体实现，需要依赖于 Chromium 项目的 AwContents 模块来实现。整体模块架构如图 17-3 所示。

AwContents 基于 Content 之上，专门针对 WebView 需求进行封装，此封装只是针对 Android 平台实现。同样道理，WebView 也是基于 ContentAPI（Web Contents 和 ContentViewCore 等）之上的，这一点同 Content Shell 和 Chromium 浏览器没有太大差异，区别在于它们对很多 Delegate 类的实现不同，这是 ContentAPI 用于让使用者参与内部逻辑和实现的过程。具体来说，主要有以下两个方面的差异。

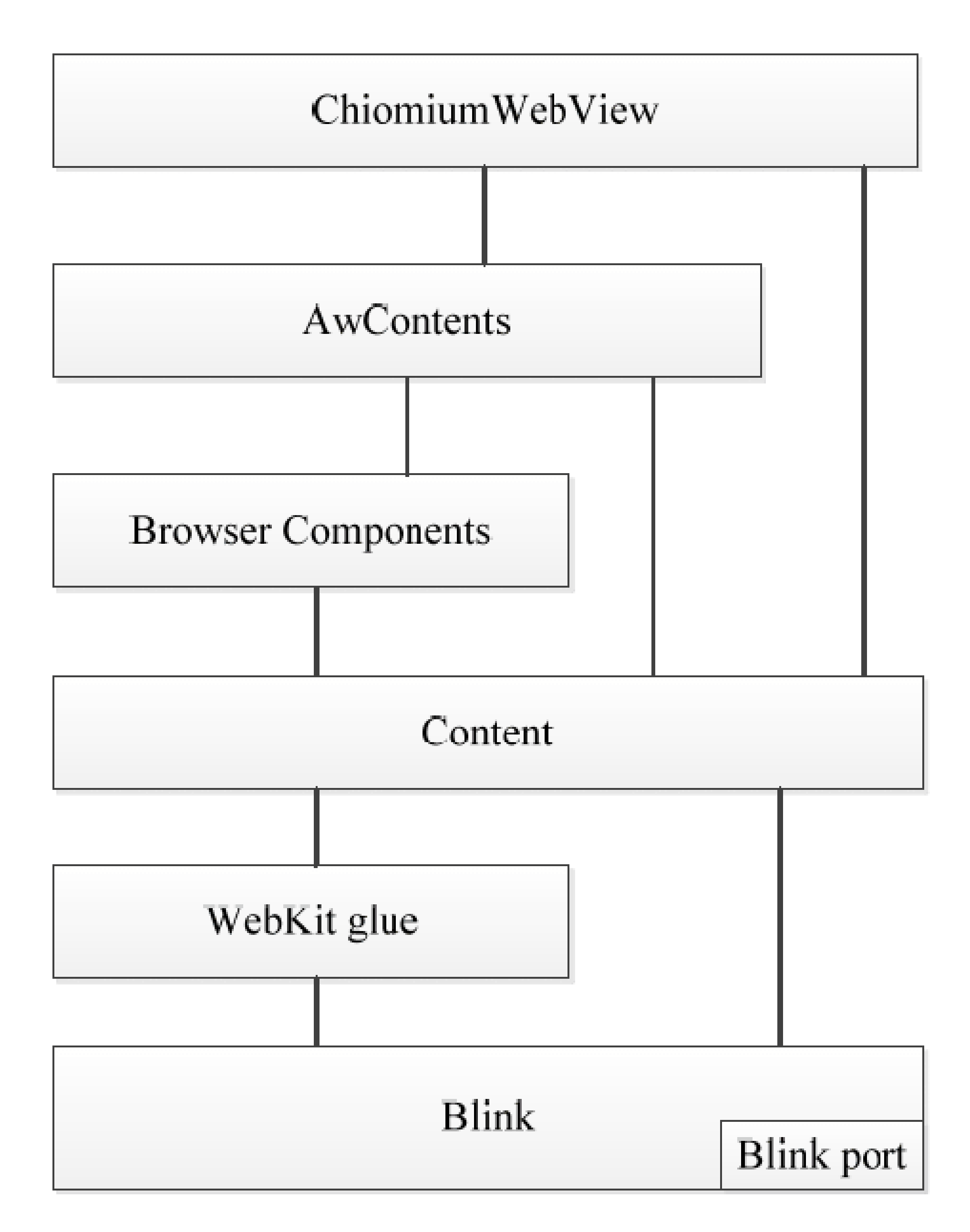


图 17-3　AwContents 模块的架构图

（1）渲染机制

因为 WebView 提供的是一个 View 控件，所以 View 控件的容器可能会接受存储在 CPU 中的结构，例如 bitmap，也可能是存储在 GPU 内存中的结构，例如 surface，所以需要提供两种不同的输出结果。

Chromium 引入了一种新的合成器 UberCompositor++，该合成器支持输出到 GPU 和 CPU 内存两种方式。对于 Compositor 的结果输出到给定 View 的 GPU 内存这种方式来说，关键点在于实现 AwContents.InternalAccessDelegate 接口的 requestDrawGL()方法。与 requestDrawGL()实现有关的代码不多，只是文件 DrawGLFunctor.java 和 GraphicsUtils.java，还有与之关联的本地实现文件 draw_gl_functor.cpp、raphic_buffer_impl.cpp 和 graphics_utils.cpp，本地代码位于 frameworks/webview/chromium/plat_support 目录下，此部分并不是只使用了 Android SDK 和 NDK 的 API，而且也涉及了 Android 源码。

（2）进程

目前 WebView 只支持单进程方式，未来版本应该支持多进程方式。单进程意味着无法使用 Android 的 isolated UID 机制，因此，从某种程度上来讲，安全性降低了，而且页面的渲染崩溃会导致使用 WebView 的应用程序崩溃。

17.2　WebView 类简介

在 Android 5.0 系统中，WebView 类的实现文件 WebView.java 位于目录 Frameworks/base/core/java/android/webkit 中。

类 WebView 继承于 AbsoluteLayout，实现 OnGlobalFocusChangeListener 和 OnHierarchyChangeListener，类 WebView 的构造函数的具体实现代码如下所示。



```
@SuppressWarnings("deprecation")
uper() call into deprecated base class constructor
    protected WebView(Context context, AttributeSet attrs, int defStyle,
            Map<String, Object> javaScriptInterfaces, boolean privateBrowsing) {
        super(context, attrs, defStyle);
        if (context == null) {
            throw new IllegalArgumentException("Invalid context argument");
        }
        sEnforceThreadChecking = context.getApplicationInfo().targetSdkVersion >=
                Build.VERSION_CODES.JELLY_BEAN_MR2;
        checkThread();
        if (DebugFlags.TRACE_API) Log.d(LOGTAG, "WebView<init>");

        ensureProviderCreated();
        mProvider.init(javaScriptInterfaces, privateBrowsing);
        CookieSyncManager.setGetInstanceIsAllowed();
    }
    public void addJavascriptInterface(Object object, String name) {
        checkThread();
        if (DebugFlags.TRACE_API) Log.d(LOGTAG, "addJavascriptInterface=" + name);
        mProvider.addJavascriptInterface(object, name);
    }
    public boolean canGoBack() {
        checkThread();
        return mProvider.canGoBack();
    }
public boolean canGoBackOrForward(int steps) {
        checkThread();
        return mProvider.canGoBackOrForward(steps);
    }
public void goForward() {
        checkThread();
        if (DebugFlags.TRACE_API) Log.d(LOGTAG, "goForward");
        mProvider.goForward();
    }
   public boolean canZoomIn() {
        checkThread();
        return mProvider.canZoomIn();
    }
    //注入所提供的 Java 对象到这个 Web 视图
    public void addJavascriptInterface(Object object, String name) {
        checkThread();
        if (DebugFlags.TRACE_API) Log.d(LOGTAG, "addJavascriptInterface=" + name);
        mProvider.addJavascriptInterface(object, name);
    }
    //获取此 WebView 是否浏览历史的项目
    public boolean canGoBack() {
        checkThread();
        return mProvider.canGoBack();
    }
    //获取页面是否前进和回退选项
```

```
    public boolean canGoBackOrForward(int steps) {
        checkThread();
        return mProvider.canGoBackOrForward(steps);
    }
    //获取此 WebView 是否有历史前进选项
    public void goForward() {
        checkThread();
        if (DebugFlags.TRACE_API) Log.d(LOGTAG, "goForward");
        mProvider.goForward();
    }
    //获取此 WebView 是否可以放大
    public boolean canZoomIn() {
        checkThread();
        return mProvider.canZoomIn();
    }
    //告诉这个 Web 视图，以清除其内部的前进/后退清单
    public void clearHistory() {
        checkThread();
        if (DebugFlags.TRACE_API) Log.d(LOGTAG, "clearHistory");
        mProvider.clearHistory();
    }
    //获取第一个子串组成的物理位置的地址
    public static String findAddress(String addr) {
        return getFactory().getStatics().findAddress(addr);
    }
    //获取 HTML 内容的高度
    public int getContentHeight() {
        checkThread();
        return mProvider.getContentHeight();
    }
    //获取当前页面的原始 URL
    public String getOriginalUrl() {
        checkThread();
        return mProvider.getOriginalUrl();
    }
    //获取当前页面的进度
    public int getProgress() {
        checkThread();
        return mProvider.getProgress();
    }
```

17.3 WebViewProvider 接口

在 Android 5.0 中，WebView 本身并不实现具体功能，而是将所有的处理工作交给 WebViewProvider 来实现。WebViewProvider 只是一个接口，最后由实现者决定采用哪种引擎。接口 WebViewProvider 的实现文件是 WebViewProvider.java，其中定义了很多接口函数，主要实现代码如下所示。

```
public interface WebViewProvider {
    public void init(Map<String, Object> javaScriptInterfaces,
```



```
            boolean privateBrowsing);
    public void setHorizontalScrollbarOverlay(boolean overlay);
    public void setVerticalScrollbarOverlay(boolean overlay);
    public boolean overlayHorizontalScrollbar();
    public boolean overlayVerticalScrollbar();
    public int getVisibleTitleHeight();
    public SslCertificate getCertificate();
    public void setCertificate(SslCertificate certificate);
    public void savePassword(String host, String username, String password);
    public void setHttpAuthUsernamePassword(String host, String realm,
            String username, String password);
    public String[] getHttpAuthUsernamePassword(String host, String realm);
    public void destroy();
    public void setNetworkAvailable(boolean networkUp);
    public WebBackForwardList saveState(Bundle outState);
    public boolean savePicture(Bundle b, final File dest);
    public boolean restorePicture(Bundle b, File src);
    public WebBackForwardList restoreState(Bundle inState);
    public void loadUrl(String url, Map<String, String> additionalHttpHeaders);
    public void loadUrl(String url);
    public void postUrl(String url, byte[] postData);
    public void loadData(String data, String mimeType, String encoding);
    public void loadDataWithBaseURL(String baseUrl, String data,
            String mimeType, String encoding, String historyUrl);
    public void evaluateJavaScript(String script, ValueCallback<String> resultCallback);
    public void saveWebArchive(String filename);
    public void saveWebArchive(String basename, boolean autoname, ValueCallback<String> callback);
    public void stopLoading();
    public void reload();
    public boolean canGoBack();
    public void goBack();
    public boolean canGoForward();
    public void goForward();
    public boolean canGoBackOrForward(int steps);
    public void goBackOrForward(int steps);
    public boolean isPrivateBrowsingEnabled();
    public boolean pageUp(boolean top);
    public boolean pageDown(boolean bottom);
    public void clearView();
    public Picture capturePicture();
    public PrintDocumentAdapter createPrintDocumentAdapter();
    public float getScale();
    public void setInitialScale(int scaleInPercent);
    public void invokeZoomPicker();
    public HitTestResult getHitTestResult();
    public void requestFocusNodeHref(Message hrefMsg);
    public void requestImageRef(Message msg);
    public String getUrl();
    public String getOriginalUrl();
    public String getTitle();
    public Bitmap getFavicon();
```

```
    public String getTouchIconUrl();
    public int getProgress();
    public int getContentHeight();
    public int getContentWidth();
    public void pauseTimers();
    public void resumeTimers();
    public void onPause();
    public void onResume();
    public boolean isPaused();
    public void freeMemory();
    public void clearCache(boolean includeDiskFiles);
    public void clearFormData();
    public void clearHistory();
    public void clearSslPreferences();
    public WebBackForwardList copyBackForwardList();
    public void setFindListener(WebView.FindListener listener);
    public void findNext(boolean forward);
    public int findAll(String find);
    public void findAllAsync(String find);
    public boolean showFindDialog(String text, boolean showIme);
    public void clearMatches();
    public void documentHasImages(Message response);
    public void setWebViewClient(WebViewClient client);
    public void setDownloadListener(DownloadListener listener);
    public void setWebChromeClient(WebChromeClient client);
    public void setPictureListener(PictureListener listener);
    public void addJavascriptInterface(Object obj, String interfaceName);
    public void removeJavascriptInterface(String interfaceName);
    public WebSettings getSettings();
    public void setMapTrackballToArrowKeys(boolean setMap);
    public void flingScroll(int vx, int vy);
    public View getZoomControls();
    public boolean canZoomIn();
    public boolean canZoomOut();
    public boolean zoomIn();
    public boolean zoomOut();
    public void dumpViewHierarchyWithProperties(BufferedWriter out, int level);
    public View findHierarchyView(String className, int hashCode);
        public boolean shouldDelayChildPressedState();
        public AccessibilityNodeProvider getAccessibilityNodeProvider();
        public void onInitializeAccessibilityNodeInfo(AccessibilityNodeInfo info);
        public void onInitializeAccessibilityEvent(AccessibilityEvent event);
        public boolean performAccessibilityAction(int action, Bundle arguments);
        public void setOverScrollMode(int mode);
        public void setScrollBarStyle(int style);
        public void onDrawVerticalScrollBar(Canvas canvas, Drawable scrollBar, int l, int t, int r, int b);
        public void onOverScrolled(int scrollX, int scrollY, boolean clampedX, boolean clampedY);
        public void onWindowVisibilityChanged(int visibility);
        public void onDraw(Canvas canvas);
        public void setLayoutParams(LayoutParams layoutParams);
        public boolean performLongClick();
```

```
        public void onConfigurationChanged(Configuration newConfig);
        public InputConnection onCreateInputConnection(EditorInfo outAttrs);
        public boolean onKeyMultiple(int keyCode, int repeatCount, KeyEvent event);
        public boolean onKeyDown(int keyCode, KeyEvent event);
...
```

在 Android 5.0 系统中，通过 WebViewChromium 实现 WebViewProvider 接口。另外，WebViewFactory 也采用了相似的结构，决定如何实例化 WebViewFactoryProvider，WebViewFactoryProvider 的关键接口是 createWebView，功能是创建具体的 WebViewProvider。

17.4　WebViewChromium 详解

在 Android 5.0 系统中，WebViewChromium 实现 WebViewProvider 接口。WebViewChromium 在文件 Frameworks/webview/chromium/java/com/android/webview/chromium/WebViewChromium.java 中定义。

在文件 WebViewChromium.java 中实现了 WebViewProvider 中各个方法的具体功能，主要实现代码如下所示。

```
...
    public void init(final Map<String, Object> javaScriptInterfaces,
            final boolean privateBrowsing) {
        if (privateBrowsing) {
            mFactory.startYourEngines(true);
            final String msg = "Private browsing is not supported in WebView.";
            if (mAppTargetSdkVersion >= Build.VERSION_CODES.KITKAT) {
                throw new IllegalArgumentException(msg);
            } else {
                Log.w(TAG, msg);
                TextView warningLabel = new TextView(mWebView.getContext());
                warningLabel.setText(mWebView.getContext().getString(
                        com.android.internal.R.string.webviewchromium_private_browsing_warning));
                mWebView.addView(warningLabel);
            }
        }

        if (mAppTargetSdkVersion >= Build.VERSION_CODES.JELLY_BEAN_MR2) {
            mFactory.startYourEngines(false);
            checkThread();
        } else if (!mFactory.hasStarted()) {
            if (Looper.myLooper() == Looper.getMainLooper()) {
                mFactory.startYourEngines(true);
            }
        }

        final boolean isAccessFromFileURLsGrantedByDefault =
                mAppTargetSdkVersion < Build.VERSION_CODES.JELLY_BEAN;
        final boolean areLegacyQuirksEnabled =
                mAppTargetSdkVersion < Build.VERSION_CODES.KITKAT;
```

```
        mContentsClientAdapter = new WebViewContentsClientAdapter(mWebView);
        mWebSettings = new ContentSettingsAdapter(new AwSettings(
                mWebView.getContext(), isAccessFromFileURLsGrantedByDefault,
                areLegacyQuirksEnabled));

        mRunQueue.addTask(new Runnable() {
                @Override
                public void run() {
                    initForReal();
                    if (privateBrowsing) {
                        // Intentionally irreversibly disable the webview instance, so that private
                        // user data cannot leak through misuse of a non-privateBrowing WebView
                        // instance. Can't just null out mAwContents as we never null-check it
                        // before use
                        destroy();
                    }
                }
        });
    }

    private void initForReal() {
        mAwContents = new AwContents(mFactory.getBrowserContext(), mWebView,
                new InternalAccessAdapter(), mContentsClientAdapter, new AwLayoutSizer(),
                mWebSettings.getAwSettings());

        if (mAppTargetSdkVersion >= Build.VERSION_CODES.KITKAT) {
            AwContents.setShouldDownloadFavicons();
        }
    }

    void startYourEngine() {
        mRunQueue.drainQueue();
    }

    private RuntimeException createThreadException() {
        return new IllegalStateException(
                "Calling View methods on another thread than the UI thread.");
    }
…
```

上面只是列出了几个方法，具体代码读者可以参阅 Android 5.0 的源代码。在 Android 5.0 系统中，WebViewChromium 通过调用 Chromium 项目的 AwContents 模块来实现具体功能。

17.5　WebViewChromiumFactoryProvider 详解

在 Android 5.0 系统中，文件 WebViewChromiumFactoryProvider.java 的功能是实例化 WebViewFactoryProvider，此文件位于目录 Frameworks/webview/chromium/java/com/android/webview/



chromium/中。

文件 WebViewChromiumFactoryProvider.java 也同样需要依赖于 Chromium 项目的 AwContents 模块来实现具体功能，主要实现代码如下所示。

```
private void ensureChromiumStartedLocked(boolean onMainThread) {
    assert Thread.holdsLock(mLock);
    if (mStarted) {
        return;
    }
    Lo//获取此 WebView 是否有浏览历史前进选项

    //获取页面是否有前进和回退选项
oper looper = !onMainThread ? Looper.myLooper() : Looper.getMainLooper();
    Log.v("WebViewChromium", "Binding Chromium to the " +
            (onMainThread ? "main":"background") + " looper " + looper);
    ThreadUtils.setUiThread(looper);
    if (ThreadUtils.runningOnUiThread()) {
        startChromiumLocked();
        return;
    }
    ThreadUtils.postOnUiThread(new Runnable() {
        @Override
        public void run() {
            synchronized (mLock) {
                startChromiumLocked();
            }
        }
    });
    while (!mStarted) {
        try {
            // Important: wait() releases |mLock| so the UI thread can take it :-)
            mLock.wait();
        } catch (InterruptedException e) {
        }
    }
}
private void startChromiumLocked() {
    assert Thread.holdsLock(mLock) && ThreadUtils.runningOnUiThread();
    mLock.notifyAll();
    if (mStarted) {
        return;
    }
    if (Build.IS_DEBUGGABLE) {
        CommandLine.initFromFile(COMMAND_LINE_FILE);
    } else {
        CommandLine.init(null);
    }
    CommandLine cl = CommandLine.getInstance();
    cl.appendSwitch("enable-dcheck");
    if (!cl.hasSwitch("disable-webview-gl-mode")) {
```

```
            cl.appendSwitch("testing-webview-gl-mode");
        }
        Context context = ActivityThread.currentApplication();
        if (context.getApplicationInfo().targetSdkVersion < Build.VERSION_CODES.KITKAT) {
            cl.appendSwitch("enable-webview-classic-workarounds");
        }
        ResourceExtractor.setMandatoryPaksToExtract("");
        try {
            LibraryLoader.ensureInitialized();
        } catch(ProcessInitException e) {
            throw new RuntimeException("Error initializing WebView library", e);
        }
        PathService.override(PathService.DIR_MODULE, "/system/lib/");
        final int DIR_RESOURCE_PAKS_ANDROID = 3003;
        PathService.override(DIR_RESOURCE_PAKS_ANDROID,
                "/system/framework/webview/paks");
        AwBrowserProcess.start(ActivityThread.currentApplication());
        initPlatSupportLibrary();
        if (Build.IS_DEBUGGABLE) {
            setWebContentsDebuggingEnabled(true);
        }
        mStarted = true;
        for (WeakReference<WebViewChromium> wvc : mWebViewsToStart) {
            WebViewChromium w = wvc.get();
            if (w != null) {
                w.startYourEngine();
            }
        }
        mWebViewsToStart.clear();
        mWebViewsToStart = null;
}
@Override
public Statics getStatics() {
        synchronized (mLock) {
            if (mStaticMethods == null) {
                ensureChromiumStartedLocked(true);
                mStaticMethods = new WebViewFactoryProvider.Statics() {
                    @Override
                    public String findAddress(String addr) {
                        return ContentViewStatics.findAddress(addr);
                    }
                    @Override
                    public void setPlatformNotificationsEnabled(boolean enable) {
                    }
                    @Override
                    public String getDefaultUserAgent(Context context) {
                       return AwSettings.getDefaultUserAgent();
                    }
                    @Override
                    public void setWebContentsDebuggingEnabled(boolean enable) {
                        if (!Build.IS_DEBUGGABLE) {
```

```
                                WebViewChromiumFactoryProvider.this.
                                        setWebContentsDebuggingEnabled(enable);
                            }
                        }
                    };
                }
            }
            return mStaticMethods;
        }
```

17.6　AwContents 架构

在 ContentAPI 之上，Chromium 的 WebView 实现封装了一个新的类 AwContents，该类主要基于 ContentViewCore 类的实现。不同的是，AwContents 需要基于一个原来存在于 chrome/目录下的模块 BrowserComponents。但是因为 AwContents 不应该依赖该目录，所以将 Chrome 中的一些浏览器模块化是 Chromium 的一个发展方向。

AwContents 模块的实现文件是 AwContents.java，在目录 External/chromium_org/android_webview/java/src/org/chromium/android_webview/中实现。

文件 AwContents.java 提供的不是 WebView 的 API，所以需要使用桥接层将 AwContents 连接到 WebView。文件 AwContents.java 的主要实现代码如下所示。

```
@JNINamespace("android_webview")
public class AwContents {
    private static final String TAG = "AwContents";

    private static final String WEB_ARCHIVE_EXTENSION = ".mht";

    private static final float ZOOM_CONTROLS_EPSILON = 0.007f;
    public static class HitTestData {
        public int hitTestResultType;
        public String hitTestResultExtraData;

        public String href;
        public String anchorText;
        public String imgSrc;
    }
    public AwContents(AwBrowserContext browserContext, ViewGroup containerView,
            InternalAccessDelegate internalAccessAdapter, AwContentsClient contentsClient,
            boolean isAccessFromFileURLsGrantedByDefault, AwLayoutSizer layoutSizer,
            boolean supportsLegacyQuirks) {
        this(browserContext, containerView, internalAccessAdapter, contentsClient,
                layoutSizer, new AwSettings(containerView.getContext(),
                        isAccessFromFileURLsGrantedByDefault, supportsLegacyQuirks));
    }

    public AwContents(AwBrowserContext browserContext, ViewGroup containerView,
            InternalAccessDelegate internalAccessAdapter, AwContentsClient contentsClient,
```

```
            AwLayoutSizer layoutSizer, AwSettings settings) {
        mBrowserContext = browserContext;
        mContainerView = containerView;
        mInternalAccessAdapter = internalAccessAdapter;
        mContentsClient = contentsClient;
        mLayoutSizer = layoutSizer;
        mSettings = settings;
        mDIPScale = DeviceDisplayInfo.create(mContainerView.getContext()).getDIPScale();
        mLayoutSizer.setDelegate(new AwLayoutSizerDelegate());
        mLayoutSizer.setDIPScale(mDIPScale);
        mWebContentsDelegate = new AwWebContentsDelegateAdapter(contentsClient, mContainerView);
        mContentsClientBridge = new AwContentsClientBridge(contentsClient);
        mZoomControls = new AwZoomControls(this);
        mIoThreadClient = new IoThreadClientImpl();
        mInterceptNavigationDelegate = new InterceptNavigationDelegateImpl();

        AwSettings.ZoomSupportChangeListener zoomListener =
                new AwSettings.ZoomSupportChangeListener() {
                    @Override
                    public void onGestureZoomSupportChanged(boolean supportsGestureZoom) {
                        mContentViewCore.updateMultiTouchZoomSupport(supportsGestureZoom);
                        mContentViewCore.updateDoubleTapDragSupport(supportsGestureZoom);
                    }

                };
        mSettings.setZoomListener(zoomListener);
        mDefaultVideoPosterRequestHandler = new DefaultVideoPosterRequestHandler(mContentsClient);
        mSettings.setDefaultVideoPosterURL(
                mDefaultVideoPosterRequestHandler.getDefaultVideoPosterURL());
        mSettings.setDIPScale(mDIPScale);
        mScrollOffsetManager = new AwScrollOffsetManager(new AwScrollOffsetManagerDelegate(),
                new OverScroller(mContainerView.getContext()));

        setOverScrollMode(mContainerView.getOverScrollMode());
        setScrollBarStyle(mInternalAccessAdapter.super_getScrollBarStyle());
        mContainerView.addOnLayoutChangeListener(new AwLayoutChangeListener());

        setNewAwContents(nativeInit(mBrowserContext));

        onVisibilityChanged(mContainerView, mContainerView.getVisibility());
        onWindowVisibilityChanged(mContainerView.getWindowVisibility());
    }

    /**
     * AwContents 实例的初始化
     *
     * TAKE CARE! This method can get called multiple times per java instance. Code accordingly.
     * See the native class declaration for more details on relative object lifetimes
     */
    private void setNewAwContents(int newAwContentsPtr) {
        if (mNativeAwContents != 0) {
```

```
                destroy();
                mContentViewCore = null;
            }

            assert mNativeAwContents == 0 && mCleanupReference == null && mContentViewCore == null;

            mNativeAwContents = newAwContentsPtr;

            mCleanupReference = new CleanupReference(this, new DestroyRunnable(mNativeAwContents));

            int nativeWebContents = nativeGetWebContents(mNativeAwContents);
            mContentViewCore = createAndInitializeContentViewCore(
                        mContainerView, mInternalAccessAdapter, nativeWebContents,
                        new AwGestureStateListener(), mContentsClient.getContentViewClient(),
                        mZoomControls);
            nativeSetJavaPeers(mNativeAwContents, this, mWebContentsDelegate, mContentsClientBridge,
                        mIoThreadClient, mInterceptNavigationDelegate);
            mContentsClient.installWebContentsObserver(mContentViewCore);
            mSettings.setWebContents(nativeWebContents);
            nativeSetDipScale(mNativeAwContents, (float) mDIPScale);
            updateGlobalVisibleRect();

            mContentViewCore.onShow();
    }

        /**
         * 呼叫"source" AwContents，打开一个弹出式窗口
         */
        public void supplyContentsForPopup(AwContents newContents) {
            int popupNativeAwContents = nativeReleasePopupAwContents(mNativeAwContents);
            if (popupNativeAwContents == 0) {
                Log.w(TAG, "Popup WebView bind failed: no pending content.");
                if (newContents != null) newContents.destroy();
                return;
            }
            if (newContents == null) {
                nativeDestroy(popupNativeAwContents);
                return;
            }

            newContents.receivePopupContents(popupNativeAwContents);
        }

        private void receivePopupContents(int popupNativeAwContents) {
            final boolean wasAttached = mIsAttachedToWindow;
            final boolean wasViewVisible = mIsViewVisible;
            final boolean wasWindowVisible = mIsWindowVisible;
            final boolean wasPaused = mIsPaused;
            final boolean wasFocused = mContainerViewFocused;
```

```
        final boolean wasWindowFocused = mWindowFocused;

        if (wasFocused) onFocusChanged(false, 0, null);
        if (wasWindowFocused) onWindowFocusChanged(false);
        if (wasViewVisible) setViewVisibilityInternal(false);
        if (wasWindowVisible) setWindowVisibilityInternal(false);
        if (!wasPaused) onPause();

        setNewAwContents(popupNativeAwContents);

        if (!wasPaused) onResume();
        if (wasAttached) onAttachedToWindow();
        onSizeChanged(mContainerView.getWidth(), mContainerView.getHeight(), 0, 0);
        if (wasWindowVisible) setWindowVisibilityInternal(true);
        if (wasViewVisible) setViewVisibilityInternal(true);
        if (wasWindowFocused) onWindowFocusChanged(wasWindowFocused);
        if (wasFocused) onFocusChanged(true, 0, null);
    }

    /**
     *删除此对象的本地副本
     */
    public void destroy() {
        if (mCleanupReference != null) {
            mContentViewCore.destroy();
            mNativeAwContents = 0;

            if (mIsAttachedToWindow) {
                if (mPendingDetachCleanupReferences == null) {
                    mPendingDetachCleanupReferences = new ArrayList<CleanupReference>();
                }
                mPendingDetachCleanupReferences.add(mCleanupReference);
            } else {
                mCleanupReference.cleanupNow();
            }
            mCleanupReference = null;
        }

        assert !mContentViewCore.isAlive();
        assert mNativeAwContents == 0;
    }
```

由此可见，AwContents 和之前版本的 WebView 是一个概念，用于存放网页渲染的结果，并且 AwContents 都是基于 Android SDK/NDK 开发，并没有使用 Android Source 中未公开的 API 和库。

17.7 实现 Mixed Content 模式

在 Android 5.0 的 WebView 系统中，将通过函数 setMixedContentMode()和 getMixedContentMode()



实现网页的 MixedContent 模式。函数 setMixedContentMode()和 getMixedContentMode()在文件 platform/frameworks/base/core/java/android/webkit/WebSettings.java 中定义，具体定义原型如下所示。

```
public abstract void setMixedContentMode(int mode);
public abstract int getMixedContentMode();
public static final int MIXED_CONTENT_ALWAYS_ALLOW = 0;
public static final int MIXED_CONTENT_NEVER_ALLOW = 1;
public static final int MIXED_CONTENT_COMPATIBILITY_MODE = 2;
```

函数 setMixedContentMode()和 getMixedContentMode()在文件 AwSettings.java 中定义，具体实现代码如下所示。

```
public void setMixedContentMode(int mode) {
    synchronized (mAwSettingsLock) {
        if (mMixedContentMode != mode) {
            mMixedContentMode = mode;
            mEventHandler.updateWebkitPreferencesLocked();
        }
    }
}

public int getMixedContentMode() {
    synchronized (mAwSettingsLock) {
        return mMixedContentMode;
    }
}
```

17.8 引入第三方 Cookie

在 Android 5.0 的 WebView 系统中，将通过函数 setAcceptThirdPartyCookies()和 acceptThirdPartyCookies()在网页中引入第三方 Cookie。函数 setAcceptThirdPartyCookies()和 acceptThirdPartyCookies()在文件 platform/frameworks/base/core/java/android/webkit/CookieManager.java 中定义，具体定义原型如下所示。

```
    public class CookieManager {
    protected CookieManager() {
    }
    @Override
    protected Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException("doesn't implement Cloneable");
    }
    public static synchronized CookieManager getInstance() {
    return WebViewFactory.getProvider().getCookieManager();
    }
    public synchronized void setAcceptCookie(boolean accept) {
    throw new MustOverrideException();
    }
    public synchronized boolean acceptCookie() {
```

```
throw new MustOverrideException();
}
public void setAcceptThirdPartyCookies(WebView webview, boolean accept) {
throw new MustOverrideException();
}
public boolean acceptThirdPartyCookies(WebView webview) {
throw new MustOverrideException();
}
public void setCookie(String url, String value) {
throw new MustOverrideException();
}
public void setCookie(String url, String value, ValueCallback<Boolean> callback) {
throw new MustOverrideException();
}
public String getCookie(String url) {
throw new MustOverrideException();
}
public String getCookie(String url, boolean privateBrowsing) {
throw new MustOverrideException();
}
public synchronized String getCookie(WebAddress uri) {
throw new MustOverrideException();
}
public void removeSessionCookie() {
throw new MustOverrideException();
}
public void removeSessionCookies(ValueCallback<Boolean> callback) {
throw new MustOverrideException();
}
@Deprecated
public void removeAllCookie() {
throw new MustOverrideException();
}
public void removeAllCookies(ValueCallback<Boolean> callback) {
throw new MustOverrideException();
}
public synchronized boolean hasCookies() {
throw new MustOverrideException();
}
public synchronized boolean hasCookies(boolean privateBrowsing) {
throw new MustOverrideException();
}
@Deprecated
public void removeExpiredCookie() {
throw new MustOverrideException();
}
public void flush() {
flushCookieStore();
}
```



```
    protected void flushCookieStore() {
    }
    public static boolean allowFileSchemeCookies() {
    return getInstance().allowFileSchemeCookiesImpl();
    }
    protected boolean allowFileSchemeCookiesImpl() {
    throw new MustOverrideException();
    }
    public static void setAcceptFileSchemeCookies(boolean accept) {
    getInstance().setAcceptFileSchemeCookiesImpl(accept);
    }
    protected void setAcceptFileSchemeCookiesImpl(boolean accept) {
    throw new MustOverrideException();
    }
}
```

第 18 章　Wi-Fi 系统架构详解

Wi-Fi 是一种可以将个人计算机、手持设备（如 PDA、手机）等终端以无线方式互相连接的技术。Wi-Fi 是一个无线网络通信技术的品牌，由 Wi-Fi 联盟（Wi-Fi Alliance）所持有，目的是改善基于 IEEE 802.11 标准的无线网络产品之间的互通性。有些用户会把 Wi-Fi 及 IEEE 802.11 混为一谈，甚至直接把 Wi-Fi 等同于无线网络，但事实并非如此。本章将简要介绍在 Android 5.0 平台中 Wi-Fi 系统的核心架构知识。

18.1　Wi-Fi 系统基础

在 Android 系统中，存在了一个无线控制模块，利用该模块可控制无线网络。打开方式如下：依次选择 Menu | Settings | Wireless$networks | Mobile network settings 命令，进入如图 18-1 所示的界面，在此界面可以选择一个移动网络。

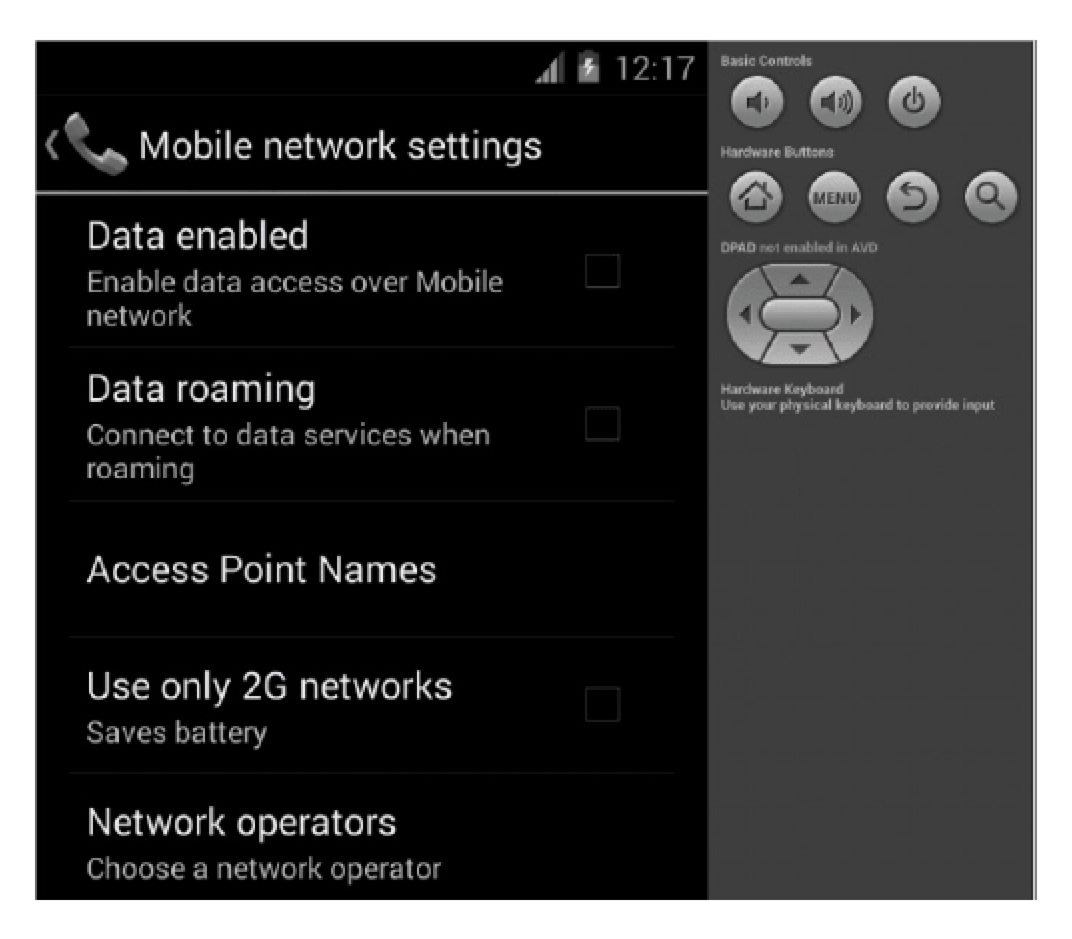


图 18-1　在此可以选择一个移动网络

Wi-Fi 系统的上层接口包括数据部分和控制部分，数据部分通常是一个和以太网卡类似的网络设备，控制部分用于实现接入点操作和安全验证处理。

在软件层，Wi-Fi 系统包括 Linux 内核程序和协议，还包括本地部分、Java 框架类。Wi-Fi 系统向 Java 应用程序层提供了控制类的接口。

Android 平台中 Wi-Fi 系统的基本层次结构如图 18-2 所示。

由图 18-1 可知，Android 平台中 Wi-Fi 系统从上到下主要包括 Java 框架类、Android 适配器库、wpa_supplicant 守护进程、驱动程序和协议，这几部分的系统结构如图 18-3 所示。

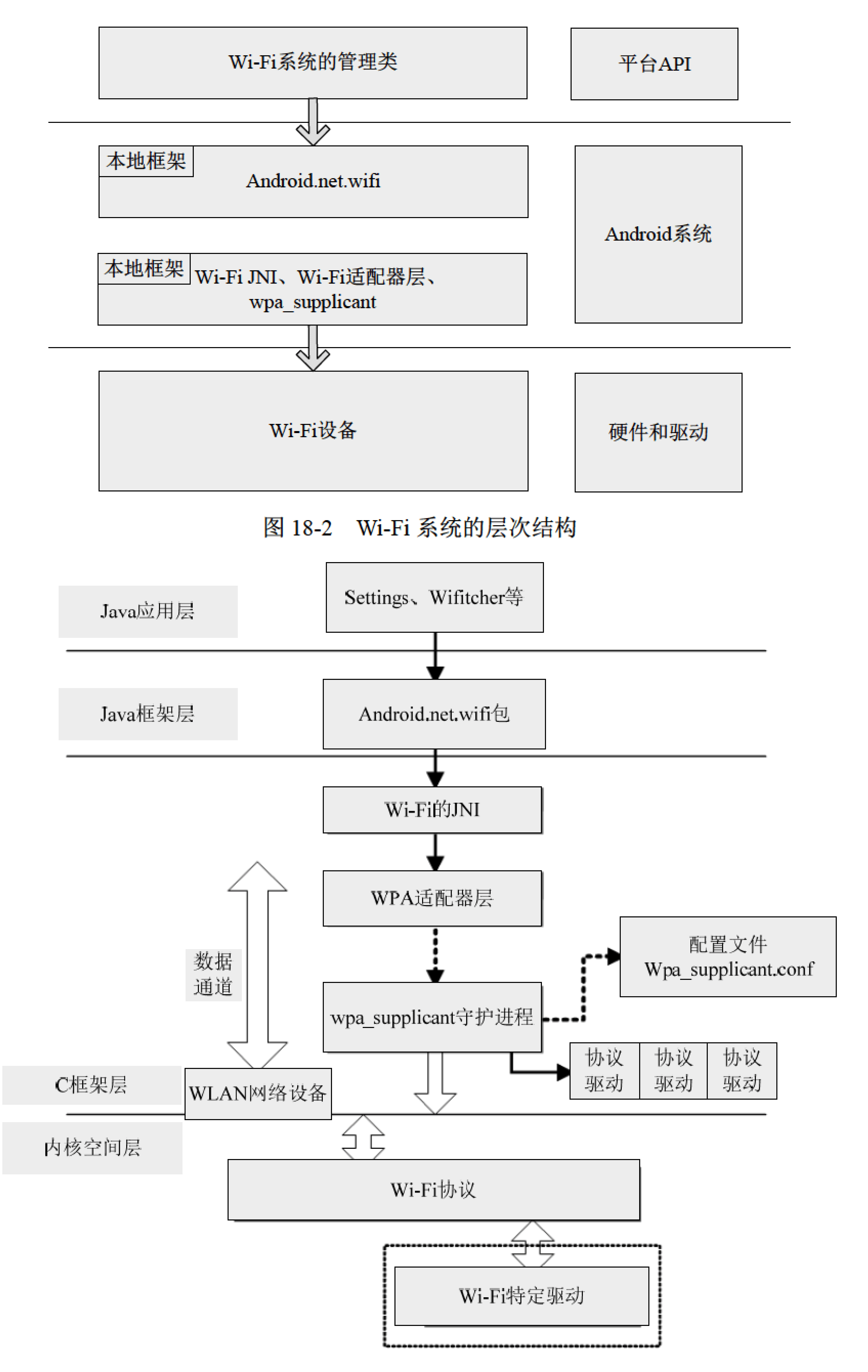


图 18-2　Wi-Fi 系统的层次结构

图 18-3　Wi-Fi 的系统结构

图 18-3 中各个部分的具体说明如下所示。

（1）Wi-Fi 用户空间的程序和库，对应路径为 external/wpa_supplicant/。

在此生成库 libwpaclient.so 和守护进程 wpa_supplicant。

（2）Wi-Fi 管理库，即适配器库，通过调用库 libwpaclient.so 成为 wpa_supplicant 在 Android 中的

客户端。对应路径为 hardware/libhardware_legary/wifi/。

（3）JNI 部分的对应路径为 frameworks/base/core/jni/android_net_wifi_Wifi.cpp。

（4）Java 框架部分的对应路径为 frameworks/base/services/java/com/android/server/和 frameworks/base/wifi/java/android/net/wifi/。

在 android.net.wifi 将作为 Android 平台的 API 供 Java 应用程序层使用。

（5）Wi-Fi Settings 应用程序的对应路径为 packages/apps/Settings/src/com/android/settings/wifi/。

注意：Android 和 Linux 的差异

先看 Wi-Fi 在 Android 中是如何工作的：Android 使用一个修改版 wpa_supplicant 作为 daemon 来控制 Wi-Fi，代码位于目录 external/wpa_supplicant 中。

wpa_supplicant 是通过 Socket 与文件 hardware/libhardware_legacy/wifi/wifi.c 进行通信。UI 通过 android.net.wifi package（frameworks/base/wifi/java/android/net/wifi/）发送命令给文件 wifi.c。相应的 JNI 实现位于文件 frameworks/base/core/jni/android_net_wifi_Wifi.cpp 中，更高一级的网络管理位于目录 frameworks/base/core/java/android/net 中。

在 Android 中的无线局域网部分是标准的系统，并且针对特定的硬件平台，所以需要移植和改动的内容并不多。在 Linux 内核中有 Wi-Fi 的标准协议，不同硬件平台的差异仅体现在 Wi-Fi 芯片驱动程序。除了这些芯片级驱动的差异外，Linux 内核中已经给出了在 Android 中实现其他无线局域网部分的具体方法。

而在 Android 用户空间中，使用了标准的 wpa_supplicant 守护进程，这也是一个标准的实现，所以无须为 Wi-Fi 增加单独的硬件抽象层代码，只需进行简单的配置工作即可。

18.2 Wi-Fi 本地部分架构

本地实现部分主要包括 wpa_supplicant 以及 wpa_supplicant 适配层。WPA（Wi-Fi Protected Access，Wi-Fi 网络安全存取）是一种基于标准的可互操作的 WLAN 安全性增强解决方案，可大大增强现有以及未来无线局域网系统的数据保护和访问控制水平。

wpa_supplicant 适配层是通用的 wpa_supplicant 的封装，在 Android 中作为 Wi-Fi 部分的硬件抽象层来使用。wpa_supplicant 适配层主要用于封装与 wpa_supplicant 守护进程的通信，以提供给 Android 框架使用。它实现了加载、控制和消息监控等功能。wpa_supplicant 适配层的头文件是 hardware/libhardware_legacy/include/hardware_legacy/wifi.h。

wpa_supplicant 的标准结构框图如图 18-4 所示。

重点关注框图的下半部分，即 wpa_supplicant 是如何与 DRIVER 进行联系的。整个过程暂以 AP 发出 SCAN 命令为主线。由于现在大部分 Wi-Fi DRIVER 都支持 wext，所以假设设备走的是 wext 这条线，其实用 ndis 也一样，整个流程也差不多。

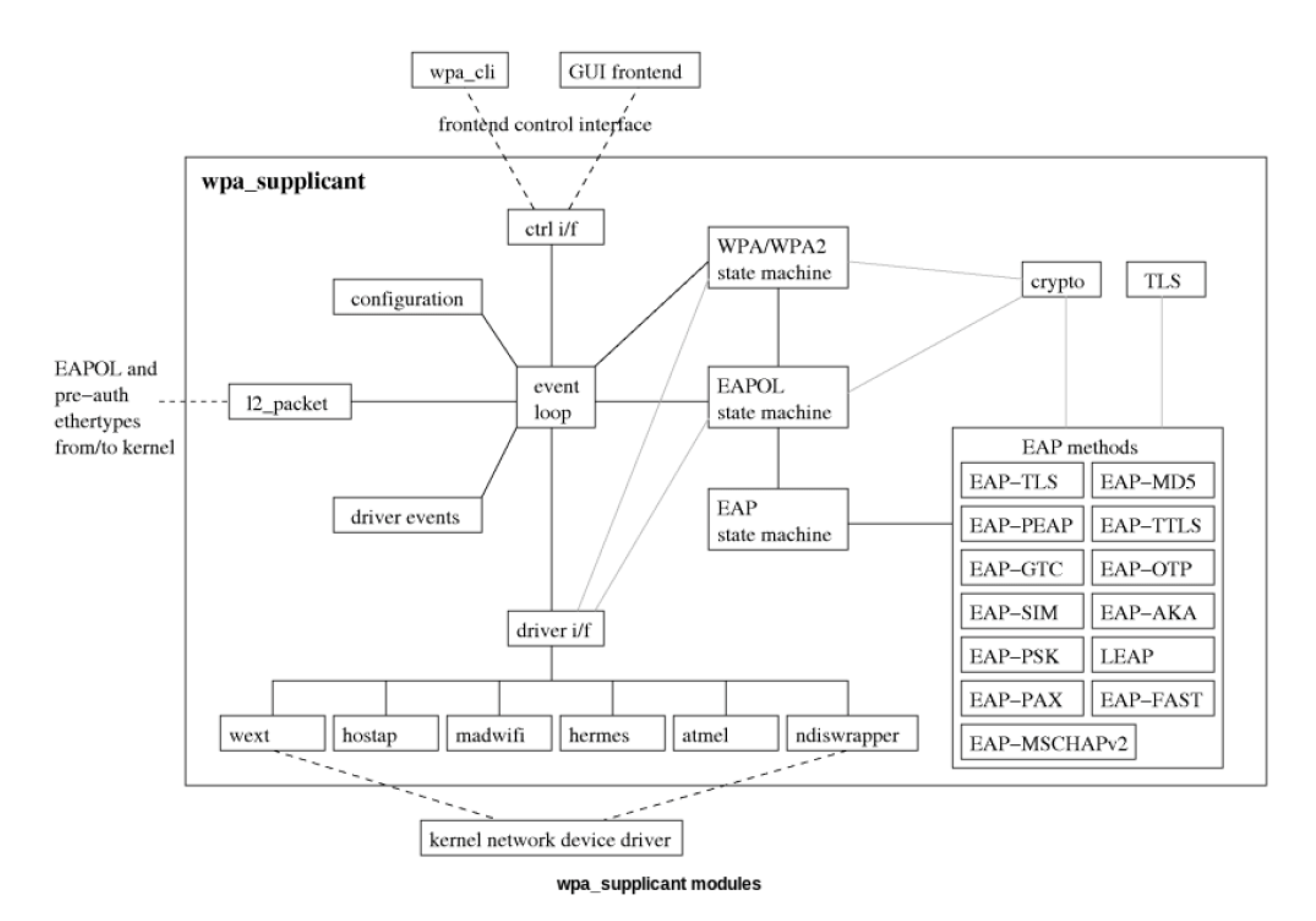


图 18-4　wpa_supplicant 的标准结构框图

首先要说的是，在文件 Driver.h 中存在一个名为 wpa_driver_ops 的结构体，这个结构体在 Driver.c 中被声明，代码如下。

```
#ifdef CONFIG_DRIVER_WEXT
extern struct wpa_driver_ops wpa_driver_wext_ops;
```

然后在文件 driver_wext.c 中填写该结构体的成员，代码如下。

```
const struct wpa_driver_ops wpa_driver_wext_ops = {
	.name = "wext",
	.desc = "Linux wireless extensions (generic)",
	.get_bssid = wpa_driver_wext_get_bssid,
	.get_ssid = wpa_driver_wext_get_ssid,
	.set_key = wpa_driver_wext_set_key,
	.set_countermeasures = wpa_driver_wext_set_countermeasures,
	.scan2 = wpa_driver_wext_scan,
	.get_scan_results2 = wpa_driver_wext_get_scan_results,
	.deauthenticate = wpa_driver_wext_deauthenticate,
	.associate = wpa_driver_wext_associate,
	.init = wpa_driver_wext_init,
	.deinit = wpa_driver_wext_deinit,
	.add_pmkid = wpa_driver_wext_add_pmkid,
	.remove_pmkid = wpa_driver_wext_remove_pmkid,
	.flush_pmkid = wpa_driver_wext_flush_pmkid,
	.get_capa = wpa_driver_wext_get_capa,
```

```
        .set_operstate = wpa_driver_wext_set_operstate,
        .get_radio_name = wext_get_radio_name,
#ifdef ANDROID
        .sched_scan = wext_sched_scan,
        .stop_sched_scan = wext_stop_sched_scan,
#endif /* ANDROID */
};
```

上述成员其实都是驱动和 wpa_supplicant 的接口，以 SCAN 为例的代码如下所示。

```
int wpa_driver_wext_scan(void *priv, const u8 *ssid, size_t ssid_len)
```

通过如下代码可以看出 wpa_cupplicant 是通过 IOCTL 来调用 SOCKET 与 DRIVER 进行通信的，并给 DRIVER 下达 SIOCSIWSCAN 命令。

```
if (ioctl(drv->ioctl_sock, SIOCSIWSCAN, &iwr) < 0)
```

这样，当一个命令从 AP 到 Framework 传到 C++本地库再到 wpa_supplicant 适配层，再由 wpa_supplicant 下 CMD 传给 DRIVER 的路线就打通了。

因为 Wi-Fi 模块是采用 SDIO 总线来控制的，所以应该先记录下 CLIENT DRIVER 的 SDIO 部分的结构，此部分的 SDIO 分为 3 层，分别是 SdioDrv、SdioAdapter 和 SdioBusDrv。其中，SdioBusDrv 是 Client Driver 中 SDIO 与 Wi-Fi 模块的接口，SdioAdapter 是 SdioDrv 和 SdioBusDrv 之间的适配层，SdioDrv 是 Client Driver 中 SDIO 与 LINUX KERNEL 中的 MMC SDIO 的接口。这 3 部分只需要关注一下 SdioDrv 即可，另外两层都只是对它的封装。

在 SdioDrv 中提供了下面的功能。

```
static struct sdio_driver tiwlan_sdio_drv = {
        .probe = tiwlan_sdio_probe,
        .remove = tiwlan_sdio_remove,
        .name = "sdio_tiwlan",
        .id_table = tiwl12xx_devices,
};
int sdioDrv_EnableFunction(unsigned int uFunc)
int sdioDrv_EnableInterrupt(unsigned int uFunc)
```

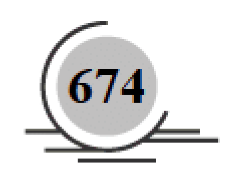
Sdio 的读写实际上调用了 MMC/Core 中的如下功能函数。

```
static int mmc_io_rw_direct_host()
```

Sdio 功能部分读者只需简单了解即可，一般 HOST 部分芯片厂商都会提供完整的解决方案。此处的主要任务还是了解 Wi-Fi 模块。

首先看 Wi-Fi 模块的入口函数 wlanDrvIf_ModuleInit()，此入口函数调用了函数 wlanDrvIf_Create()，主要代码如下所示。

```
static int wlanDrvIf_Create (void)
{
        TWlanDrvIfObj *drv; //这个结构体为代表设备，包含 Linux 网络设备结构体 net_device
        pDrvStaticHandle = drv;
        drv->pWorkQueue = create_singlethread_workqueue (TIWLAN_DRV_NAME);//创建了工作队列
```



```
    rc = wlanDrvIf_SetupNetif (drv);
    drv->wl_sock = netlink_kernel_create( NETLINK_USERSOCK, 0, NULL, NULL, THIS_MODULE );
    //创建了接收 wpa_supplicant 的 SOCKET 接口
    rc = drvMain_Create (drv,
                        &drv->tCommon.hDrvMain,
                        &drv->tCommon.hCmdHndlr,
                        &drv->tCommon.hContext,
                        &drv->tCommon.hTxDataQ,
                        &drv->tCommon.hTxMgmtQ,
                        &drv->tCommon.hTxCtrl,
                        &drv->tCommon.hTWD,
                        &drv->tCommon.hEvHandler,
                        &drv->tCommon.hCmdDispatch,
                        &drv->tCommon.hReport,
                        &drv->tCommon.hPwrState);
    rc = hPlatform_initInterrupt (drv, (void*)wlanDrvIf_HandleInterrupt);
   return 0;
}
```

在调用完函数 wlanDrvIf_Create()后，初始化 Wi-Fi 模块的工作就完成了。接下来开始分析如何实现初始化。首先分析函数 wlanDrvIf_SetupNetif(drv)，其主要实现代码如下所示。

```
static int wlanDrvIf_SetupNetif (TWlanDrvIfObj *drv)
{
   struct net_device *dev;
   int res;
   dev = alloc_etherdev (0);          //开始申请 Linux 网络设备
   if (dev == NULL)
     ether_setup (dev);               //开始建立网络接口 ，这两个都是 Linux 网络设备驱动的标准函数
   dev->netdev_ops = &wlan_netdev_ops;
   wlanDrvWext_Init (dev);
   res = register_netdev (dev);
   hPlatform_SetupPm(wlanDrvIf_Suspend, wlanDrvIf_Resume, pDrvStaticHandle);
}
```

在此初始化了 wlanDrvWext_Inti(dev)，接下来需要注册网络设备 dev，在 wlan_netdev_ops 中的定义代码如下所示。

```
static const struct net_device_ops wlan_netdev_ops = {
        .ndo_open = wlanDrvIf_Open,
        .ndo_stop = wlanDrvIf_Release,
        .ndo_do_ioctl = NULL,
        .ndo_start_xmit = wlanDrvIf_Xmit,
        .ndo_get_stats = wlanDrvIf_NetGetStat,
        .ndo_validate_addr = NULL,
};
```

上述代码名字对应的都是 Linux 网络设备驱动的命令字，最后需要调用 rc=drvMain_CreateI，通过此函数完成相关模块的初始化工作。

18.3 Wi-Fi JNI 部分架构

在 Android 系统中，Wi-Fi 系统的 JNI 部分实现的源码文件为 frameworks/base/core/jni/android_net_wifi_Wifi.cpp。

JNI 层的接口注册到 Java 层的源代码文件为 frameworks/base/wifi/java/android/net/wifi/WifiNative.java。

WifiNative 将为 WifiService、WifiStateTracker、WifiMonitor 等几个 Wi-Fi 框架内部组件提供底层操作支持。

此处实现的本地函数都是通过调用 wpa_supplicant 适配层的接口来实现的（包含适配层的头文件 wifi.h）。wpa_supplicant 适配层是通用的 wpa_supplicant 的封装，在 Android 中作为 Wi-Fi 部分的硬件抽象层来使用。wpa_supplicant 适配层主要用于封装与 wpa_supplicant 守护进程的通信，以提供给 Android 框架使用。它实现了加载、控制和消息监控等功能。wpa_supplicant 适配层的头文件为 hardware/libhardware_legacy/include/hardware_legacy/wifi.h。

文件 wifi.h 是 Wi-Fi 适配器层对 JNI 部分的接口，其中包含了一些加载和连接的控制接口，主要包括如下两个接口。

- ☑ wifi_command()：负责将命令发送到 Wi-Fi 下层。
- ☑ wifi_wait_for_event()：负责事件进入通道，此函数将被阻塞，一直到收到一个 Wi-Fi 事件为止，并且以字符串的形式返回。

在文件 wifi.h 中定义上述接口的代码如下所示。

```
int wifi_command(const char *command, char *reply, size_t *reply_len);
int wifi_wait_for_event(char *buf, size_t len);
```

在文件 wifi.c 中实现了上述两个接口，具体代码如下所示。

```
int wifi_command(const char *command, char *reply, size_t *reply_len)
{
    return wifi_send_command(ctrl_conn, command, reply, reply_len);
}
int wifi_wait_for_event(char *buf, size_t buflen)
{
    size_t nread = buflen - 1;
    int fd;
    fd_set rfds;
    int result;
    struct timeval tval;
    struct timeval *tptr;

    if (monitor_conn == NULL)
        return 0;

    result = wpa_ctrl_recv(monitor_conn, buf, &nread);
    if (result < 0) {
        LOGD("wpa_ctrl_recv failed: %s\n", strerror(errno));
        return -1;
```

```
    }
    buf[nread] = '\0';
    if (result == 0 && nread == 0) {
        /* Fabricate an event to pass up */
        LOGD("Received EOF on supplicant socket\n");
        strncpy(buf, WPA_EVENT_TERMINATING " - signal 0 received", buflen-1);
        buf[buflen-1] = '\0';
        return strlen(buf);
    }
    if (buf[0] == '<') {
        char *match = strchr(buf, '>');
        if (match != NULL) {
            nread -= (match+1-buf);
            memmove(buf, match+1, nread+1);
        }
    }
    return nread;
}
```

18.4　Java FrameWork 部分的源码

Wi-Fi 系统 Java 部分的核心是根据 IWifiManager 接口所创建的 Binder 服务器端和客户端，服务器端是 WifiService，客户端是 WifiManager。具体结构如图 18-5 所示。

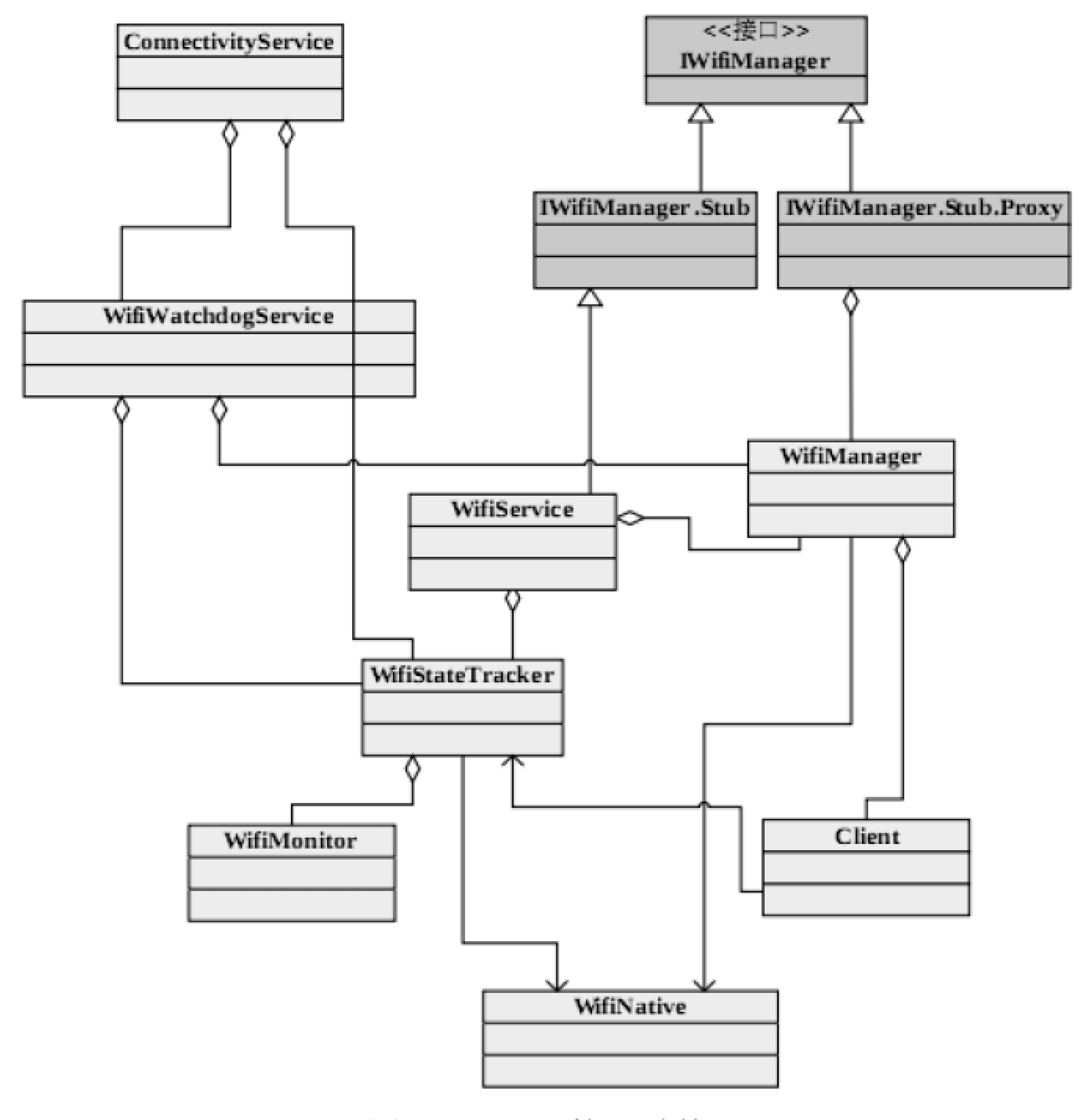


图 18-5　JNI 接口结构

Wi-Fi 系统的 Java 部分代码实现的目录如下。

☑ frameworks/base/wifi/java/android/net/：Wi-Fi 服务层的内容。

☑ frameworks/base/services/java/com/android/server/wifi：Wi-Fi 部分的接口。

编译 IWifiManager.aidl 生成文件 IWifiManager.java，并生成 IWifiManager.Stub（服务器端抽象类）和 IWifiManager.Stub.Proxy（客户端代理实现类）。WifiService 通过继承 IWifiManager.Stub 实现，而客户端通过 getService()函数获取 IWifiManager.Stub.Proxy（即 Service 的代理类），将其作为参数传递给 WifiManager，供其与 WifiService 通信时使用。

18.4.1 WifiManager 详解

WifiManager 部分表示 Wi-Fi 与外界的接口，用户通过它来访问 Wi-Fi 的核心功能。WifiWatchdogService 这一系统组件也是用 WifiManager 来执行一些具体操作。文件 WifiManager.java 的主要实现代码如下所示。

```
public int addNetwork(WifiConfiguration config) {
	if (config == null) {
		return -1;
	}
	config.networkId = -1;
	return addOrUpdateNetwork(config);
}
public int updateNetwork(WifiConfiguration config) {
	if (config == null || config.networkId < 0) {
		return -1;
	}
	return addOrUpdateNetwork(config);
}
private int addOrUpdateNetwork(WifiConfiguration config) {
	try {
		return mService.addOrUpdateNetwork(config);
	} catch (RemoteException e) {
		return -1;
	}
}
public boolean removeNetwork(int netId) {
	try {
		return mService.removeNetwork(netId);
	} catch (RemoteException e) {
		return false;
	}
}
public boolean enableNetwork(int netId, boolean disableOthers) {
	try {
		return mService.enableNetwork(netId, disableOthers);
	} catch (RemoteException e) {
		return false;
	}
}
public boolean disableNetwork(int netId) {
```

```
        try {
            return mService.disableNetwork(netId);
        } catch (RemoteException e) {
            return false;
        }
}
public boolean disconnect() {
        try {
            mService.disconnect();
            return true;
        } catch (RemoteException e) {
            return false;
        }
}
public boolean reconnect() {
        try {
            mService.reconnect();
            return true;
        } catch (RemoteException e) {
            return false;
        }
}
public boolean reassociate() {
        try {
            mService.reassociate();
            return true;
        } catch (RemoteException e) {
            return false;
        }
}
public boolean pingSupplicant() {
        if (mService == null)
            return false;
        try {
            return mService.pingSupplicant();
        } catch (RemoteException e) {
            return false;
        }
}
public boolean startScan() {
        try {
            mService.startScan();
            return true;
        } catch (RemoteException e) {
            return false;
        }
}
```

18.4.2　WifiService 详解

WifiService 部分是服务器端的实现，作为 Wi-Fi 的核心，处理实际的驱动加载、扫描、连接、断

开等命令，以及底层上报的事件。对于主动的命令控制，Wi-Fi 是一个简单的封装，针对来自客户端的控制命令，调用相应的 WifiNative 底层实现。文件 WifiService.java 的主要实现代码如下所示。

```
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case AsyncChannel.CMD_CHANNEL_HALF_CONNECTED: {
                    if (msg.arg1 == AsyncChannel.STATUS_SUCCESSFUL) {
                        if (DBG) Slog.d(TAG, "New client listening to asynchronous messages");
                        mTrafficPoller.addClient(msg.replyTo);
                    } else {
                        Slog.e(TAG, "Client connection failure, error=" + msg.arg1);
                    }
                    break;
                }
                case AsyncChannel.CMD_CHANNEL_DISCONNECTED: {
                    if (msg.arg1 == AsyncChannel.STATUS_SEND_UNSUCCESSFUL) {
                        if (DBG) Slog.d(TAG, "Send failed, client connection lost");
                    } else {
                        if (DBG) Slog.d(TAG, "Client connection lost with reason: " + msg.arg1);
                    }
                    mTrafficPoller.removeClient(msg.replyTo);
                    break;
                }
                case AsyncChannel.CMD_CHANNEL_FULL_CONNECTION: {
                    AsyncChannel ac = new AsyncChannel();
                    ac.connect(mContext, this, msg.replyTo);
                    break;
                }
                /* Client commands are forwarded to state machine */
                case WifiManager.CONNECT_NETWORK:
                case WifiManager.SAVE_NETWORK:
                case WifiManager.FORGET_NETWORK:
                case WifiManager.START_WPS:
                case WifiManager.CANCEL_WPS:
                case WifiManager.DISABLE_NETWORK:
                case WifiManager.RSSI_PKTCNT_FETCH: {
                    mWifiStateMachine.sendMessage(Message.obtain(msg));
                    break;
                }
                default: {
                    Slog.d(TAG, "ClientHandler.handleMessage ignoring msg=" + msg);
                    break;
                }
            }
        }
}
private void noteScanStart() {
        WorkSource scanWorkSource = null;
        synchronized (WifiService.this) {
            if (mScanWorkSource != null) {
```

```
                return;
            }
            scanWorkSource = new WorkSource(Binder.getCallingUid());
            mScanWorkSource = scanWorkSource;
        }

        long id = Binder.clearCallingIdentity();
        try {
            mBatteryStats.noteWifiScanStartedFromSource(scanWorkSource);
        } catch (RemoteException e) {
            Log.w(TAG, e);
        } finally {
            Binder.restoreCallingIdentity(id);
        }
    }
    private void noteScanEnd() {
        WorkSource scanWorkSource = null;
        synchronized (WifiService.this) {
            scanWorkSource = mScanWorkSource;
            mScanWorkSource = null;
        }
        if (scanWorkSource != null) {
            try {
                mBatteryStats.noteWifiScanStoppedFromSource(scanWorkSource);
            } catch (RemoteException e) {
                Log.w(TAG, e);
            }
        }
    }
    public void checkAndStartWifi() {
        /* Check if wi-fi needs to be enabled */
        boolean wifiEnabled = mSettingsStore.isWifiToggleEnabled();
        Slog.i(TAG, "WifiService starting up with Wi-Fi " +
                (wifiEnabled ? "enabled" : "disabled"));

        if (wifiEnabled) setWifiEnabled(wifiEnabled);

        mWifiWatchdogStateMachine = WifiWatchdogStateMachine.
               makeWifiWatchdogStateMachine(mContext);

    }
    public boolean removeNetwork(int netId) {
        enforceChangePermission();
        if (mWifiStateMachineChannel != null) {
            return mWifiStateMachine.syncRemoveNetwork(mWifiStateMachineChannel, netId);
        } else {
            Slog.e(TAG, "mWifiStateMachineChannel is not initialized");
            return false;
        }
    }
    public boolean enableNetwork(int netId, boolean disableOthers) {
```

```
        enforceChangePermission();
        if (mWifiStateMachineChannel != null) {
            return mWifiStateMachine.syncEnableNetwork(mWifiStateMachineChannel, netId,
                    disableOthers);
        } else {
            Slog.e(TAG, "mWifiStateMachineChannel is not initialized");
            return false;
        }
}
```

当接收到客户端的命令后，一般会将其转换成对应的自身消息塞入消息队列中，以便客户端的调用可以及时返回，然后在 WifiHandler 的 handleMessage()中处理对应的消息。而底层上报的事件，WifiService 则通过启动 WifiStateTracker 来负责处理。WifiStateTracker 和 WifiMonitor 的具体功能如下所示。

☑ WifiStateTracker：除了负责 Wi-Fi 的电源管理模式等功能外，其核心是 WifiMonitor 所实现的事件轮询机制，以及消息处理函数 handleMessage()。文件 WifiStateTracker.java 在 frameworks/base/wifi/java/android/net/wifi/目录中定义，主要实现代码如下所示。

```
public void startMonitoring(Context context, Handler target) {
        mCsHandler = target;
        mContext = context;

        mWifiManager = (WifiManager) mContext.getSystemService(Context.WIFI_SERVICE);
        IntentFilter filter = new IntentFilter();
        filter.addAction(WifiManager.NETWORK_STATE_CHANGED_ACTION);
        filter.addAction(WifiManager.LINK_CONFIGURATION_CHANGED_ACTION);

        mWifiStateReceiver = new WifiStateReceiver();
        mContext.registerReceiver(mWifiStateReceiver, filter);
}
        public void onReceive(Context context, Intent intent) {

            if (intent.getAction().equals(WifiManager.NETWORK_STATE_CHANGED_ACTION)) {
                mNetworkInfo = (NetworkInfo) intent.getParcelableExtra(
                        WifiManager.EXTRA_NETWORK_INFO);
                mLinkProperties = intent.getParcelableExtra(
                        WifiManager.EXTRA_LINK_PROPERTIES);
                if (mLinkProperties == null) {
                    mLinkProperties = new LinkProperties();
                }
                mLinkCapabilities = intent.getParcelableExtra(
                        WifiManager.EXTRA_LINK_CAPABILITIES);
                if (mLinkCapabilities == null) {
                    mLinkCapabilities = new LinkCapabilities();
                }
                NetworkInfo.State state = mNetworkInfo.getState();
                if (mLastState == state &&
                        mNetworkInfo.getDetailedState() != DetailedState.CAPTIVE_PORTAL_CHECK) {
                    return;
                } else {
                   mLastState = state;
```

```
            }
            Message msg = mCsHandler.obtainMessage(EVENT_STATE_CHANGED,
                    new NetworkInfo(mNetworkInfo));
            msg.sendToTarget();
        } else if (intent.getAction().equals(WifiManager.LINK_CONFIGURATION_CHANGED_ACTION)) {
            mLinkProperties = (LinkProperties) intent.getParcelableExtra(
                    WifiManager.EXTRA_LINK_PROPERTIES);
            Message msg = mCsHandler.obtainMessage(EVENT_CONFIGURATION_CHANGED, mNetworkInfo);
            msg.sendToTarget();
        }
    }
}
```

由此可见，WifiStateTracker 也是 Wi-Fi 部分与外界的接口，它不像 WifiManager 那样直接被实例化来操作，而是通过 Intent 机制来发消息通知给客户端注册的 BroadcastReceiver，以完成和客户端的接口。

☑ WifiMonitor：通过开启一个 MonitorThread 来实现事件的轮询，轮询的关键函数是前面提到的阻塞式函数 WifiNative.waitForEvent()。获取事件后，WifiMonitor 通过一系列的 Handler 通知给 WifiStateTracker。这里 WifiMonitor 的通知机制是将底层事件转换成 WifiStateTracker 所能识别的消息，塞入 WifiStateTracker 的消息循环中，最终在 handleMessage()中由 WifiStateTracker 完成对应的处理。文件 WifiMonitor.java 在目录 frameworks/base/wifi/java/android/net/wifi/中定义，主要实现代码如下所示。

```
public class WifiMonitor {

    private static final String TAG = "WifiMonitor";
    private static final int CONNECTED = 1;
    private static final int DISCONNECTED = 2;
    private static final int STATE_CHANGE = 3;
    private static final int SCAN_RESULTS = 4;
    private static final int LINK_SPEED = 5;
    private static final int TERMINATING = 6;
    private static final int DRIVER_STATE = 7;
    private static final int EAP_FAILURE = 8;
    private static final int UNKNOWN = 9;

    private static final String EVENT_PREFIX_STR = "CTRL-EVENT-";
    private static final int EVENT_PREFIX_LEN_STR = EVENT_PREFIX_STR.length();

    private static final String WPA_EVENT_PREFIX_STR = "WPA:";
    private static final String PASSWORD_MAY_BE_INCORRECT_STR =
        "pre-shared key may be incorrect";

    private static final String WPS_SUCCESS_STR = "WPS-SUCCESS";

    private static final String WPS_FAIL_STR = "WPS-FAIL";
    private static final String WPS_FAIL_PATTERN =
            "WPS-FAIL msg=\\d+(?: config_error=(\\d+))?(?: reason=(\\d+))?";

    private static final int CONFIG_MULTIPLE_PBC_DETECTED = 12;
```

```
private static final int CONFIG_AUTH_FAILURE = 18;

/* reason code values for reason=%d */
private static final int REASON_TKIP_ONLY_PROHIBITED = 1;
private static final int REASON_WEP_PROHIBITED = 2;

private static final String WPS_OVERLAP_STR = "WPS-OVERLAP-DETECTED";
private static final String WPS_TIMEOUT_STR = "WPS-TIMEOUT";

private static final String CONNECTED_STR = "CONNECTED";
private static final String DISCONNECTED_STR = "DISCONNECTED";
private static final String STATE_CHANGE_STR = "STATE-CHANGE";
private static final String SCAN_RESULTS_STR = "SCAN-RESULTS";
private static final String LINK_SPEED_STR = "LINK-SPEED";
private static final String TERMINATING_STR = "TERMINATING";
private static final String DRIVER_STATE_STR = "DRIVER-STATE";
private static final String EAP_FAILURE_STR = "EAP-FAILURE";
private static final String EAP_AUTH_FAILURE_STR = "EAP authentication failed";

private static Pattern mConnectedEventPattern =
    Pattern.compile("((?:[0-9a-f]{2}:){5}[0-9a-f]{2}) .* \\[id=([0-9]+) ");

private static final String P2P_EVENT_PREFIX_STR = "P2P";

private static final String P2P_DEVICE_FOUND_STR = "P2P-DEVICE-FOUND";
private static final String P2P_DEVICE_LOST_STR = "P2P-DEVICE-LOST";
private static final String P2P_FIND_STOPPED_STR = "P2P-FIND-STOPPED";
private static final String P2P_GO_NEG_REQUEST_STR = "P2P-GO-NEG-REQUEST";

private static final String P2P_GO_NEG_SUCCESS_STR = "P2P-GO-NEG-SUCCESS";
private static final String P2P_GO_NEG_FAILURE_STR = "P2P-GO-NEG-FAILURE";

private static final String P2P_GROUP_FORMATION_SUCCESS_STR =
        "P2P-GROUP-FORMATION-SUCCESS";

private static final String P2P_GROUP_FORMATION_FAILURE_STR =
        "P2P-GROUP-FORMATION-FAILURE";

private static final String P2P_GROUP_STARTED_STR = "P2P-GROUP-STARTED";
private static final String P2P_GROUP_REMOVED_STR = "P2P-GROUP-REMOVED";
private static final String P2P_INVITATION_RECEIVED_STR = "P2P-INVITATION-RECEIVED";
private static final String P2P_INVITATION_RESULT_STR = "P2P-INVITATION-RESULT";
private static final String P2P_PROV_DISC_PBC_REQ_STR = "P2P-PROV-DISC-PBC-REQ";
private static final String P2P_PROV_DISC_PBC_RSP_STR = "P2P-PROV-DISC-PBC-RESP";
private static final String P2P_PROV_DISC_ENTER_PIN_STR = "P2P-PROV-DISC-ENTER-PIN";
private static final String P2P_PROV_DISC_SHOW_PIN_STR = "P2P-PROV-DISC-SHOW-PIN";
private static final String P2P_PROV_DISC_FAILURE_STR = "P2P-PROV-DISC-FAILURE";
private static final String P2P_SERV_DISC_RESP_STR = "P2P-SERV-DISC-RESP";

private static final String HOST_AP_EVENT_PREFIX_STR = "AP";
private static final String AP_STA_CONNECTED_STR = "AP-STA-CONNECTED";
```

```
private static final String AP_STA_DISCONNECTED_STR = "AP-STA-DISCONNECTED";

private final StateMachine mStateMachine;
private final WifiNative mWifiNative;
private static final int BASE = Protocol.BASE_WIFI_MONITOR;
public static final int SUP_CONNECTION_EVENT = BASE + 1;
public static final int SUP_DISCONNECTION_EVENT = BASE + 2;
public static final int NETWORK_CONNECTION_EVENT = BASE + 3;
/* Network disconnection completed */
public static final int NETWORK_DISCONNECTION_EVENT = BASE + 4;
/* Scan results are available */
public static final int SCAN_RESULTS_EVENT = BASE + 5;
/* Supplicate state changed */
public static final int SUPPLICANT_STATE_CHANGE_EVENT = BASE + 6;
/* Password failure and EAP authentication failure */
public static final int AUTHENTICATION_FAILURE_EVENT = BASE + 7;
/* WPS success detected */
public static final int WPS_SUCCESS_EVENT = BASE + 8;
/* WPS failure detected */
public static final int WPS_FAIL_EVENT = BASE + 9;
 /* WPS overlap detected */
public static final int WPS_OVERLAP_EVENT = BASE + 10;
 /* WPS timeout detected */
public static final int WPS_TIMEOUT_EVENT = BASE + 11;
/* Driver was hung */
public static final int DRIVER_HUNG_EVENT = BASE + 12;

/* P2P events */
public static final int P2P_DEVICE_FOUND_EVENT = BASE + 21;
public static final int P2P_DEVICE_LOST_EVENT = BASE + 22;
public static final int P2P_GO_NEGOTIATION_REQUEST_EVENT = BASE + 23;
public static final int P2P_GO_NEGOTIATION_SUCCESS_EVENT = BASE + 25;
public static final int P2P_GO_NEGOTIATION_FAILURE_EVENT = BASE + 26;
public static final int P2P_GROUP_FORMATION_SUCCESS_EVENT = BASE + 27;
public static final int P2P_GROUP_FORMATION_FAILURE_EVENT = BASE + 28;
public static final int P2P_GROUP_STARTED_EVENT = BASE + 29;
public static final int P2P_GROUP_REMOVED_EVENT = BASE + 30;
public static final int P2P_INVITATION_RECEIVED_EVENT = BASE + 31;
public static final int P2P_INVITATION_RESULT_EVENT = BASE + 32;
public static final int P2P_PROV_DISC_PBC_REQ_EVENT = BASE + 33;
public static final int P2P_PROV_DISC_PBC_RSP_EVENT = BASE + 34;
public static final int P2P_PROV_DISC_ENTER_PIN_EVENT = BASE + 35;
public static final int P2P_PROV_DISC_SHOW_PIN_EVENT = BASE + 36;
public static final int P2P_FIND_STOPPED_EVENT = BASE + 37;
public static final int P2P_SERV_DISC_RESP_EVENT = BASE + 38;
public static final int P2P_PROV_DISC_FAILURE_EVENT = BASE + 39;

/* hostap events */
public static final int AP_STA_DISCONNECTED_EVENT = BASE + 41;
public static final int AP_STA_CONNECTED_EVENT = BASE + 42;
```

```
private static final String MONITOR_SOCKET_CLOSED_STR = "connection closed";
private static final String WPA_RECV_ERROR_STR = "recv error";

private int mRecvErrors = 0;
private static final int MAX_RECV_ERRORS = 10;

public WifiMonitor(StateMachine wifiStateMachine, WifiNative wifiNative) {
    mStateMachine = wifiStateMachine;
    mWifiNative = wifiNative;
}

public void startMonitoring() {
    new MonitorThread().start();
}
    public void run() {
        if (connectToSupplicant()) {
            mStateMachine.sendMessage(SUP_CONNECTION_EVENT);
        } else {
            mStateMachine.sendMessage(SUP_DISCONNECTION_EVENT);
            return;
        }

        for (;;) {
            String eventStr = mWifiNative.waitForEvent();

            if (false && eventStr.indexOf(SCAN_RESULTS_STR) == -1) {
                Log.d(TAG, "Event [" + eventStr + "]");
            }
            if (!eventStr.startsWith(EVENT_PREFIX_STR)) {
                if (eventStr.startsWith(WPA_EVENT_PREFIX_STR) &&
                        0 < eventStr.indexOf(PASSWORD_MAY_BE_INCORRECT_STR)) {
                    mStateMachine.sendMessage(AUTHENTICATION_FAILURE_EVENT);
                } else if (eventStr.startsWith(WPS_SUCCESS_STR)) {
                    mStateMachine.sendMessage(WPS_SUCCESS_EVENT);
                } else if (eventStr.startsWith(WPS_FAIL_STR)) {
                    handleWpsFailEvent(eventStr);
                } else if (eventStr.startsWith(WPS_OVERLAP_STR)) {
                    mStateMachine.sendMessage(WPS_OVERLAP_EVENT);
                } else if (eventStr.startsWith(WPS_TIMEOUT_STR)) {
                    mStateMachine.sendMessage(WPS_TIMEOUT_EVENT);
                } else if (eventStr.startsWith(P2P_EVENT_PREFIX_STR)) {
                    handleP2pEvents(eventStr);
                } else if (eventStr.startsWith(HOST_AP_EVENT_PREFIX_STR)) {
                    handleHostApEvents(eventStr);
                }
                continue;
            }

            String eventName = eventStr.substring(EVENT_PREFIX_LEN_STR);
            int nameEnd = eventName.indexOf(' ');
            if (nameEnd != -1)
```

```
        eventName = eventName.substring(0, nameEnd);
    if (eventName.length() == 0) {
        if (false) Log.i(TAG, "Received wpa_supplicant event with empty event name");
        continue;
    }
    /*
     * Map event name into event enum
     */
    int event;
    if (eventName.equals(CONNECTED_STR))
        event = CONNECTED;
    else if (eventName.equals(DISCONNECTED_STR))
        event = DISCONNECTED;
    else if (eventName.equals(STATE_CHANGE_STR))
        event = STATE_CHANGE;
    else if (eventName.equals(SCAN_RESULTS_STR))
        event = SCAN_RESULTS;
    else if (eventName.equals(LINK_SPEED_STR))
        event = LINK_SPEED;
    else if (eventName.equals(TERMINATING_STR))
        event = TERMINATING;
    else if (eventName.equals(DRIVER_STATE_STR))
        event = DRIVER_STATE;
    else if (eventName.equals(EAP_FAILURE_STR))
        event = EAP_FAILURE;
    else
        event = UNKNOWN;

    String eventData = eventStr;
    if (event == DRIVER_STATE || event == LINK_SPEED)
        eventData = eventData.split(" ")[1];
    else if (event == STATE_CHANGE || event == EAP_FAILURE) {
        int ind = eventStr.indexOf(" ");
        if (ind != -1) {
            eventData = eventStr.substring(ind + 1);
        }
    } else {
        int ind = eventStr.indexOf(" - ");
        if (ind != -1) {
            eventData = eventStr.substring(ind + 3);
        }
    }

    if (event == STATE_CHANGE) {
        handleSupplicantStateChange(eventData);
    } else if (event == DRIVER_STATE) {
        handleDriverEvent(eventData);
    } else if (event == TERMINATING) {
        /**
         * Close the supplicant connection if we see
         * too many recv errors
```

```
            */
            if (eventData.startsWith(WPA_RECV_ERROR_STR)) {
                if (++mRecvErrors > MAX_RECV_ERRORS) {
                    if (false) {
                        Log.d(TAG, "too many recv errors, closing connection");
                    }
                } else {
                    continue;
                }
            }

            mStateMachine.sendMessage(SUP_DISCONNECTION_EVENT);
            break;
        } else if (event == EAP_FAILURE) {
            if (eventData.startsWith(EAP_AUTH_FAILURE_STR)) {
                mStateMachine.sendMessage(AUTHENTICATION_FAILURE_EVENT);
            }
        } else {
            handleEvent(event, eventData);
        }
        mRecvErrors = 0;
    }
}

private boolean connectToSupplicant() {
    int connectTries = 0;

    while (true) {
        if (mWifiNative.connectToSupplicant()) {
            return true;
        }
        if (connectTries++ < 5) {
            nap(1);
        } else {
            break;
        }
    }
    return false;
}
```

18.4.3 WifiWatchdogService 详解

此部分是 ConnectivityService 所启动的服务，但它并不是通过 Binder 来实现的服务。其作用是监控同一个网络内的接入点（Access Point），如果当前接入点的 DNS 无法 ping 通，就自动切换到下一个接入点。WifiWatchdogService 通过 WifiManager 和 WifiStateTracker 辅助完成具体的控制动作。在 WifiWatchdogService 初始化时，通过 registerForWifiBroadcasts 注册获取网络变化的 BroadcastReceiver，也就是捕获 WifiStateTracker 所发出的通知消息，并开启一个 WifiWatchdogThread 线程来处理获取的消息。通过更改 Setting.Secure.WIFI_WATCHDOG_ON 的配置，可以开启和关闭 WifiWatchdogService。

18.5　Setting 设置架构

Android 的 Settings 应用程序对 Wi-Fi 的使用，是典型的 Wi-Fi 应用方式，也是用户可见的 Android Wi-Fi 管理程序。此部分源代码的目录为 packages/apps/Settings/src/com/android/settings/wifi/。

Setting 中的 Wi-Fi 部分是用户可见的设置界面，提供 Wi-Fi 开关、扫描 AP、连接、断开的基本功能。其中，WifiEnabler 通过 WifiManager 来完成实际的功能，也同样注册一个 BroadcastReceiver 来响应 WifiStateTracker 所发出的通知消息。WifiEnabler 其实是一个比较简单的类，提供开启和关闭 Wi-Fi 的功能，设置外层 Wi-Fi 开关菜单，就是直接通过它来做到的。文件 WifiEnabler.java 的具体实现代码如下所示。

```
public class WifiEnabler implements CompoundButton.OnCheckedChangeListener   {
    private final Context mContext;
    private Switch mSwitch;
    private AtomicBoolean mConnected = new AtomicBoolean(false);

    private final WifiManager mWifiManager;
    private boolean mStateMachineEvent;
    private final IntentFilter mIntentFilter;
    private final BroadcastReceiver mReceiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
            String action = intent.getAction();
            if (WifiManager.WIFI_STATE_CHANGED_ACTION.equals(action)) {
                handleWifiStateChanged(intent.getIntExtra(
                        WifiManager.EXTRA_WIFI_STATE, WifiManager.WIFI_STATE_UNKNOWN));
            } else if (WifiManager.SUPPLICANT_STATE_CHANGED_ACTION.equals(action)) {
                if (!mConnected.get()) {
                    handleStateChanged(WifiInfo.getDetailedStateOf((SupplicantState)
                            intent.getParcelableExtra(WifiManager.EXTRA_NEW_STATE)));
                }
            } else if (WifiManager.NETWORK_STATE_CHANGED_ACTION.equals(action)) {
                NetworkInfo info = (NetworkInfo) intent.getParcelableExtra(
                        WifiManager.EXTRA_NETWORK_INFO);
                mConnected.set(info.isConnected());
                handleStateChanged(info.getDetailedState());
            }
        }
    };

    public WifiEnabler(Context context, Switch switch_) {
        mContext = context;
        mSwitch = switch_;

        mWifiManager = (WifiManager) context.getSystemService(Context.WIFI_SERVICE);
        mIntentFilter = new IntentFilter(WifiManager.WIFI_STATE_CHANGED_ACTION);
        mIntentFilter.addAction(WifiManager.SUPPLICANT_STATE_CHANGED_ACTION);
        mIntentFilter.addAction(WifiManager.NETWORK_STATE_CHANGED_ACTION);
```

```
    }

    public void resume() {
        mContext.registerReceiver(mReceiver, mIntentFilter);
        mSwitch.setOnCheckedChangeListener(this);
    }

    public void pause() {
        mContext.unregisterReceiver(mReceiver);
        mSwitch.setOnCheckedChangeListener(null);
    }

    public void setSwitch(Switch switch_) {
        if (mSwitch == switch_) return;
        mSwitch.setOnCheckedChangeListener(null);
        mSwitch = switch_;
        mSwitch.setOnCheckedChangeListener(this);

        final int wifiState = mWifiManager.getWifiState();
        boolean isEnabled = wifiState == WifiManager.WIFI_STATE_ENABLED;
        boolean isDisabled = wifiState == WifiManager.WIFI_STATE_DISABLED;
        mSwitch.setChecked(isEnabled);
        mSwitch.setEnabled(isEnabled || isDisabled);
    }

    public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
        if (mStateMachineEvent) {
            return;
        }
        if (isChecked && !WirelessSettings.isRadioAllowed(mContext, Settings.Global.RADIO_WIFI)) {
            Toast.makeText(mContext, R.string.wifi_in_airplane_mode, Toast.LENGTH_SHORT).show();
            buttonView.setChecked(false);
        }

        int wifiApState = mWifiManager.getWifiApState();
        if (isChecked && ((wifiApState == WifiManager.WIFI_AP_STATE_ENABLING) ||
                (wifiApState == WifiManager.WIFI_AP_STATE_ENABLED))) {
            mWifiManager.setWifiApEnabled(null, false);
        }

        if (mWifiManager.setWifiEnabled(isChecked)) {
            mSwitch.setEnabled(false);
        } else {
            Toast.makeText(mContext, R.string.wifi_error, Toast.LENGTH_SHORT).show();
        }
    }

    private void handleWifiStateChanged(int state) {
        switch (state) {
```

```
                case WifiManager.WIFI_STATE_ENABLING:
                    mSwitch.setEnabled(false);
                    break;
                case WifiManager.WIFI_STATE_ENABLED:
                    setSwitchChecked(true);
                    mSwitch.setEnabled(true);
                    break;
                case WifiManager.WIFI_STATE_DISABLING:
                    mSwitch.setEnabled(false);
                    break;
                case WifiManager.WIFI_STATE_DISABLED:
                    setSwitchChecked(false);
                    mSwitch.setEnabled(true);
                    break;
                default:
                    setSwitchChecked(false);
                    mSwitch.setEnabled(true);
                    break;
            }
        }

        private void setSwitchChecked(boolean checked) {
            if (checked != mSwitch.isChecked()) {
                mStateMachineEvent = true;
                mSwitch.setChecked(checked);
                mStateMachineEvent = false;
            }
        }

        private void handleStateChanged(@SuppressWarnings("unused") NetworkInfo.DetailedState state) {
            if (state != null && mSwitch.isChecked()) {
                WifiInfo info = mWifiManager.getConnectionInfo();
                if (info != null) {
                    //setSummary(Summary.get(mContext, info.getSSID(), state));
                }
            }
            */
        }
}
```

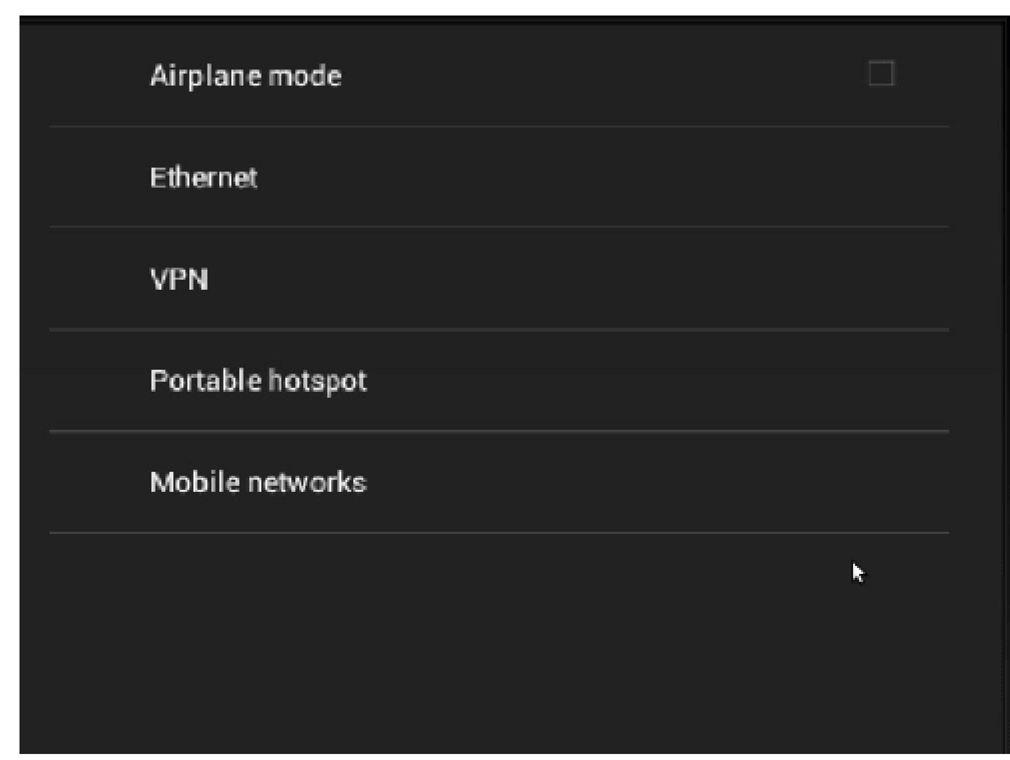
在 Android 系统 Setting 界面的 wireless 配置项中会看到 Portable Wi-Fi hotspot 和 Configure Wi-Fi hotspot setting 选项，在此可以配置 AP 的名称、加密方式和密码等，如图 18-6 所示。

当完成上述设置后系统开始接受响应，从此开启了整个 Android SoftAP 的启动序幕。首先通过文件 packages/apps/Settings/src/com/android/settings/TetherSettings.java 的 onPreferenceChange()函数接收到 Softap 状态改变信息，具体代码如下所示。

```
public boolean onPreferenceChange(Preference preference, Object value) {
    boolean enable = (Boolean) value;

    if (enable) {
```

```
            startProvisioningIfNecessary(WIFI_TETHERING);
        } else {
            mWifiApEnabler.setSoftapEnabled(false);
        }
        return false;
}
```



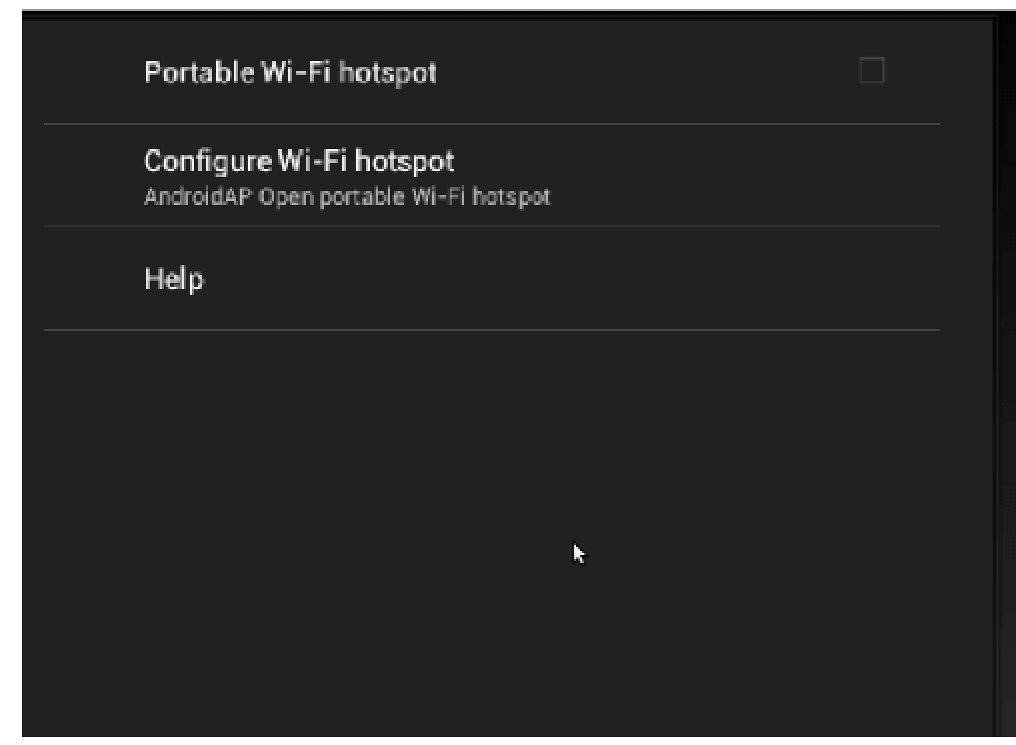




图 18-6　Portable Wi-Fi hotspot 和 Configure Wi-Fi hotspot setting 选项

当 Softap 开启时 enable 为真，因而执行 startProvisioningIfNecessary(WIFI_TETHERING)，Softap 开启时 enable 为真，因而执行 startProvisioningIfNecessary(WIFI_TETHERING)。

```
private void startProvisioningIfNecessary(int choice) {
        mTetherChoice = choice;
        if (isProvisioningNeeded()) {
            Intent intent = new Intent(Intent.ACTION_MAIN);
            intent.setClassName(mProvisionApp[0], mProvisionApp[1]);
            startActivityForResult(intent, PROVISION_REQUEST);
        } else {
            startTethering();
        }
}
```

在上述代码中，isProvisioningNeeded()用来检测是否需要进行一些准备工作。如果无须准备工作则执行 startTethering()函数，具体实现代码如下所示。

```
private void startTethering() {
        switch (mTetherChoice) {
            case WIFI_TETHERING:
                mWifiApEnabler.setSoftapEnabled(true);
                break;
```



```
            case BLUETOOTH_TETHERING:
                BluetoothAdapter adapter = BluetoothAdapter.getDefaultAdapter();
                if (adapter.getState() == BluetoothAdapter.STATE_OFF) {
                    mBluetoothEnableForTether = true;
                    adapter.enable();
                    mBluetoothTether.setSummary(R.string.bluetooth_turning_on);
                    mBluetoothTether.setEnabled(false);
                } else {
                    BluetoothPan bluetoothPan = mBluetoothPan.get();
                    if (bluetoothPan != null) bluetoothPan.setBluetoothTethering(true);
                    mBluetoothTether.setSummary(R.string.bluetooth_tethering_available_subtext);
                }
                break;
            case USB_TETHERING:
                setUsbTethering(true);
                break;
            default:
                break;
        }
}
```

在上述代码中，因为 mTetherChoice == WIFI_TETHERING，所以继而执行文件 WiFiApEnable.java 中的 setSoftapEnabled(true)函数，具体实现代码如下所示。

```
public void setSoftapEnabled(boolean enable) {
    final ContentResolver cr = mContext.getContentResolver();
    /**
    * Disable Wifi if enabling tethering
    */
    int wifiState = mWifiManager.getWifiState(); //获取当前 Wi-Fi 的状态，如果开启则关闭且保存状态信息到变量中
    if (enable && ((wifiState == WifiManager.WIFI_STATE_ENABLING) ||
                (wifiState == WifiManager.WIFI_STATE_ENABLED))) {
        mWifiManager.setWifiEnabled(false);
        Settings.Global.putInt(cr, Settings.Global.WIFI_SAVED_STATE, 1);
    }

    if (mWifiManager.setWifiApEnabled(null, enable)) {
        /* Disable here, enabled on receiving success broadcast */
        mCheckBox.setEnabled(false);
    } else {
        mCheckBox.setSummary(R.string.wifi_error);
    }

    /**
    *   If needed, restore Wifi on tether disable
    */
    if (!enable) {
        int wifiSavedState = 0;
        try {
            wifiSavedState = Settings.Global.getInt(cr, Settings.Global.WIFI_SAVED_STATE);
        } catch (Settings.SettingNotFoundException e) {
```

```
                ;
            }
            if (wifiSavedState == 1) {
                mWifiManager.setWifiEnabled(true);
                Settings.Global.putInt(cr, Settings.Global.WIFI_SAVED_STATE, 0);
            }
        }
}
```

在上述代码中，首先检测 Wi-Fi 当前状态。如果正在打开或者已经打开，则关闭 Wi-Fi 并将此状态记录下来，以便关闭 Softap 时能自动恢复到之前打开 Wi-Fi 的状态。在此调用文件 frameworks/base/wifi/java/android/net/wifi/WifiManager.java 中的 mWifiManager.setWifiApEnabled(null,enable)函数。

具体实现代码如下所示。

```
public boolean setWifiApEnabled(WifiConfiguration wifiConfig, boolean enabled) {
    try {
        mService.setWifiApEnabled(wifiConfig, enabled);
        return true;
    } catch (RemoteException e) {
        return false;
    }
}
```

然后转向服务层中的文件 frameworks/base/services/java/com/android/server/WifiService.java 中。

通过函数 setWifiApEnabled()调用最基础也是最重要的 Wi-Fi 状态机中的 setWifiApEnabled 实例，具体实现代码如下所示。

```
public void setWifiApEnabled(WifiConfiguration wifiConfig, boolean enabled) {
    enforceChangePermission();
    mWifiStateMachine.setWifiApEnabled(wifiConfig, enabled);
}
```

另外，文件 WifiStatusTest.java 用于接收不同的 Intent，具体实现代码如下所示。

```
private final BroadcastReceiver mWifiStateReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        if (intent.getAction().equals(WifiManager.WIFI_STATE_CHANGED_ACTION)) {
            handleWifiStateChanged(intent.getIntExtra(WifiManager.EXTRA_WIFI_STATE,
                        WifiManager.WIFI_STATE_UNKNOWN));
        } else if (intent.getAction().equals(WifiManager.NETWORK_STATE_CHANGED_ACTION)) {
            handleNetworkStateChanged(
                    (NetworkInfo) intent.getParcelableExtra(WifiManager.EXTRA_NETWORK_INFO));
        } else if (intent.getAction().equals(WifiManager.SCAN_RESULTS_AVAILABLE_ACTION)) {
            handleScanResultsAvailable();
        } else if (intent.getAction().equals(WifiManager.SUPPLICANT_CONNECTION_CHANGE_ACTION)) {
            /* TODO: handle supplicant connection change later */
        } else if (intent.getAction().equals(WifiManager.SUPPLICANT_STATE_CHANGED_ACTION)) {
            handleSupplicantStateChanged(
```

```
                    (SupplicantState) intent.getParcelableExtra(WifiManager.EXTRA_NEW_STATE),
                    intent.hasExtra(WifiManager.EXTRA_SUPPLICANT_ERROR),
                    intent.getIntExtra(WifiManager.EXTRA_SUPPLICANT_ERROR, 0));
        } else if (intent.getAction().equals(WifiManager.RSSI_CHANGED_ACTION)) {
            handleSignalChanged(intent.getIntExtra(WifiManager.EXTRA_NEW_RSSI, 0));
        } else if (intent.getAction().equals(WifiManager.NETWORK_IDS_CHANGED_ACTION)) {
            /* TODO: handle network id change info later */
        } else {
            Log.e(TAG, "Received an unknown Wifi Intent");
        }
    }
};
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    mWifiManager = (WifiManager) getSystemService(WIFI_SERVICE);

    mWifiStateFilter = new IntentFilter(WifiManager.WIFI_STATE_CHANGED_ACTION);
    mWifiStateFilter.addAction(WifiManager.NETWORK_STATE_CHANGED_ACTION);
    mWifiStateFilter.addAction(WifiManager.SCAN_RESULTS_AVAILABLE_ACTION);
    mWifiStateFilter.addAction(WifiManager.SUPPLICANT_STATE_CHANGED_ACTION);
    mWifiStateFilter.addAction(WifiManager.RSSI_CHANGED_ACTION);
    mWifiStateFilter.addAction(WifiManager.WIFI_STATE_CHANGED_ACTION);

    registerReceiver(mWifiStateReceiver, mWifiStateFilter);

    setContentView(R.layout.wifi_status_test);

    updateButton = (Button) findViewById(R.id.update);
    updateButton.setOnClickListener(updateButtonHandler);

    mWifiState = (TextView) findViewById(R.id.wifi_state);
    mNetworkState = (TextView) findViewById(R.id.network_state);
    mSupplicantState = (TextView) findViewById(R.id.supplicant_state);
    mRSSI = (TextView) findViewById(R.id.rssi);
    mBSSID = (TextView) findViewById(R.id.bssid);
    mSSID = (TextView) findViewById(R.id.ssid);
    mHiddenSSID = (TextView) findViewById(R.id.hidden_ssid);
    mIPAddr = (TextView) findViewById(R.id.ipaddr);
    mMACAddr = (TextView) findViewById(R.id.macaddr);
    mNetworkId = (TextView) findViewById(R.id.networkid);
    mLinkSpeed = (TextView) findViewById(R.id.link_speed);
    mScanList = (TextView) findViewById(R.id.scan_list);


    mPingIpAddr = (TextView) findViewById(R.id.pingIpAddr);
    mPingHostname = (TextView) findViewById(R.id.pingHostname);
    mHttpClientTest = (TextView) findViewById(R.id.httpClientTest);

    pingTestButton = (Button) findViewById(R.id.ping_test);
    pingTestButton.setOnClickListener(mPingButtonHandler);
```

```
}
OnClickListener updateButtonHandler = new OnClickListener() {
    public void onClick(View v) {
        final WifiInfo wifiInfo = mWifiManager.getConnectionInfo();

        setWifiStateText(mWifiManager.getWifiState());
        mBSSID.setText(wifiInfo.getBSSID());
        mHiddenSSID.setText(String.valueOf(wifiInfo.getHiddenSSID()));
        int ipAddr = wifiInfo.getIpAddress();
        StringBuffer ipBuf = new StringBuffer();
        ipBuf.append(ipAddr   & 0xff).append('.').
            append((ipAddr >>>= 8) & 0xff).append('.').
            append((ipAddr >>>= 8) & 0xff).append('.').
            append((ipAddr >>>= 8) & 0xff);

        mIPAddr.setText(ipBuf);
        mLinkSpeed.setText(String.valueOf(wifiInfo.getLinkSpeed())+" Mbps");
        mMACAddr.setText(wifiInfo.getMacAddress());
        mNetworkId.setText(String.valueOf(wifiInfo.getNetworkId()));
        mRSSI.setText(String.valueOf(wifiInfo.getRssi()));
        mSSID.setText(wifiInfo.getSSID());

        SupplicantState supplicantState = wifiInfo.getSupplicantState();
        setSupplicantStateText(supplicantState);
    }
};

private void setSupplicantStateText(SupplicantState supplicantState) {
    if(SupplicantState.FOUR_WAY_HANDSHAKE.equals(supplicantState)) {
        mSupplicantState.setText("FOUR WAY HANDSHAKE");
    } else if(SupplicantState.ASSOCIATED.equals(supplicantState)) {
        mSupplicantState.setText("ASSOCIATED");
    } else if(SupplicantState.ASSOCIATING.equals(supplicantState)) {
        mSupplicantState.setText("ASSOCIATING");
    } else if(SupplicantState.COMPLETED.equals(supplicantState)) {
        mSupplicantState.setText("COMPLETED");
    } else if(SupplicantState.DISCONNECTED.equals(supplicantState)) {
        mSupplicantState.setText("DISCONNECTED");
    } else if(SupplicantState.DORMANT.equals(supplicantState)) {
        mSupplicantState.setText("DORMANT");
    } else if(SupplicantState.GROUP_HANDSHAKE.equals(supplicantState)) {
        mSupplicantState.setText("GROUP HANDSHAKE");
    } else if(SupplicantState.INACTIVE.equals(supplicantState)) {
        mSupplicantState.setText("INACTIVE");
    } else if(SupplicantState.INVALID.equals(supplicantState)) {
        mSupplicantState.setText("INVALID");
    } else if(SupplicantState.SCANNING.equals(supplicantState)) {
        mSupplicantState.setText("SCANNING");
    } else if(SupplicantState.UNINITIALIZED.equals(supplicantState)) {
        mSupplicantState.setText("UNINITIALIZED");
    } else {
```

```
            mSupplicantState.setText("BAD");
            Log.e(TAG, "supplicant state is bad");
        }
}

private void setWifiStateText(int wifiState) {
        String wifiStateString;
        switch(wifiState) {
            case WifiManager.WIFI_STATE_DISABLING:
                wifiStateString = getString(R.string.wifi_state_disabling);
                break;
            case WifiManager.WIFI_STATE_DISABLED:
                wifiStateString = getString(R.string.wifi_state_disabled);
                break;
            case WifiManager.WIFI_STATE_ENABLING:
                wifiStateString = getString(R.string.wifi_state_enabling);
                break;
            case WifiManager.WIFI_STATE_ENABLED:
                wifiStateString = getString(R.string.wifi_state_enabled);
                break;
            case WifiManager.WIFI_STATE_UNKNOWN:
                wifiStateString = getString(R.string.wifi_state_unknown);
                break;
            default:
                wifiStateString = "BAD";
                Log.e(TAG, "wifi state is bad");
                break;
        }

        mWifiState.setText(wifiStateString);
}
```